```
// Computer Program Listing Appendix Under 37 CFR 1.52(e)
// DllEntry.cpp
// Copyright (c) 2004. Zone Labs, LLC All Rights Reserved.
/* File DllEntry.cpp:
   Entry point from the OS
   A normal DLL implicitly uses C RTL function _DllMainCRTStartup, but
    this leaves the static object constructors and destructors without
    SEH protection.  Evidently, Win2K (and probably other OSes) ignores
    the SetUnhandledExceptionFilter setting inside DllMain.  We compen-
    sate by putting the C RTL's function inside a SEH block.
   Another reason not to use an unhandled exception filter is that they
    pop up in debuggers, unless the debugger is specially configured to
    ignore, and this would be a nuisance for our developers.
*/
// Windows header files
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
// Application header files
#define _NOTINCLUDE_FASTSTR
#include <VSInit.h>
#include "SecurePrivate.h"
// External functions
extern "C"  {
BOOL WINAPI _DllMainCRTStartup( HANDLE hDllHandle
                              , DWORD dwReason
                              , LPVOID lpReserved
                              ) ;
}
/* function DllEntry:
   DLL entry point
*/
extern "C"
BOOL WINAPI ZDllEntry( HANDLE hDllHandle, DWORD dwReason, LPVOID lpReser
ved)
{
  BOOL fRetCode ;
  __try  {
    // call the C RTL's DllMain, which does some work, then calls our
    //  application's DllMain
    fRetCode = _DllMainCRTStartup( hDllHandle, dwReason, lpReserved) ;
  }  // __try
  __except ( DefaultExceptionFilterEx( PEXCEPTION_POINTERS( _exception_i
nfo())))  {
    // return code, in case the filter returns EXCEPTION_EXECUTE_HANDLER
    fRetCode = FALSE ;
    // force the linker to include our self-validation code
    //  This function itself does nothing.  The file containing it has
    //   a static initializer that raises an exception that is handled
    //   by the self-validation code in VSInit.dll.
    TellLinkerToIncludeSelfValidation() ;
  }  // __except
  // return to the caller
  return fRetCode ;
}  // DllEntry
// SecurePrivate.h
// Copyright (c) 2004. Zone Labs, LLC All Rights Reserved.
/* File SecurePrivate.h:
   Internal definitions for SecureLink.lib functions
*/
#pragma once
// Windows header files
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
// Data types
typedef bool ( * PFNSTATICINITIALIZER)() ;
// Global functions
extern "C"  {
void __cdecl SecureStaticLink() ;
```

```
void __cdecl TellLinkerToIncludeSelfValidation() ;
}                                                   // extern "C"
// StaticLinks.cpp
// Copyright (c) 2004. Zone Labs, LLC All Rights Reserved.
/* File StaticLinks.cpp:
   Function for a bootstrap patching process
*/
// Windows header files
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
// Application header files
#include <VSInit.h>
#include "SecurePrivate.h"
/* function SecureStaticLink:
   Placeholder for some very obscure processing
   The static import funnels call here.  During self-validation, a
    function in VSInit patches the self-referential jump here to jump
    to a function in VSInit instead.  The first call from a funnel goes
    to that function, which in turn back patches the funnel to change
    a constant pushed and to call the static link resolver, also in
    VSInit.
   This function is linked into every self-validating module because
    of a reference in ValidateSelf().  If the module has no secure
    static links, this code is a small, harmless appendage.
*/
extern "C" __declspec( naked) void __cdecl SecureStaticLink()
{
  // we emit a 5-byte jump to the next instruction
  // Self-validation changes this code to a jump to a function in
  //  VSInit.
  __asm __emit 0xe9                             // jump ahead, until patched
  __asm __emit 0x00
  __asm __emit 0x00
  __asm __emit 0x00
  __asm __emit 0x00
  RaiseException( ERROR_SECURE_NO_PATCH, EXCEPTION_NONCONTINUABLE, 0, 0)
 ;
  // ensure the self-validation code is linked, so that this code is
  //  patched before it is executed
  TellLinkerToIncludeSelfValidation() ;
}  // SecureStaticLink
// ValidateSelf.cpp
// Copyright (c) 2004. Zone Labs, LLC All Rights Reserved.
/* File ValidateSelf:
   Call the self-validation function in VSInit.dll
*/
// Windows header files
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
// Application header files
#include <VSInit.h>
#include "SecurePrivate.h"
// Specify the initialization order
// A ZA function may call a secure import in an initializer.  We must
//  therefore ensure our static objects are constructed first.
#pragma warning ( disable : 4073)
#pragma init_seg( lib)
// Static initializer
static bool ValidateSelf() ;
static bool fInitialized = ValidateSelf() ;
/* function TellLinkerToIncludeSelfValidation:
   Validate this module
   This function does nothing, but referencing it causes the linker
    to bring in this file, which has a static initializer that
    initiates self-validation in the lib group.  Self-validating DLLs
    call this function from their custom DllEntry function.  EXEs can
    call it from somewhere in their code, or they can specify the
    function name in an /include clause on the linker command line.
```

```
   The code that actually performs self-validation is in VSInit.dll.
*/
__declspec( naked)
void __cdecl TellLinkerToIncludeSelfValidation()
{
}  // TellLinkerToIncludeSelfValidation
// All functions below are private to this file
/* function ValidateSelf:
   Perform self-validation
   The function does not return if validation fails.
*/
static bool ValidateSelf()
{
  // perform self-validation
  // The validation is done by VSInit.dll, which gets control via a
  //  continuable exception.
  // An attacker can NOP this call, but that will leave some important
  //  control blocks uninitialized.  The right place to attack is
  //  function IsPEFileValid in VSInit.dll.
  VALIDATESELF valSelf ;
  __try  {
    valSelf.dwVersion                = VALSELF_VERSION_1 ;
    valSelf.EIPInCaller              = PVOID( ValidateSelf) ;
    valSelf.dwAddrPatchResolveStatic = DWORD( SecureStaticLink) ;
    DWORD dwArg = DWORD( &valSelf) ;
    RaiseException( TRICKY_SELF_VALIDATE, 0, 1, &dwArg) ;
  }  // __try
  __except ( DefaultExceptionFilterEx( PEXCEPTION_POINTERS( _exception_i
nfo())))  {
  }  // __except
  // successful return
  return true ;
}  // ValidateSelf
// BackPatch.cpp
// Copyright (c) 2004. Zone Labs, LLC All Rights Reserved.
/* File BackPatch.cpp:
   Validate an unambiguous instruction pattern, then patch an
    instruction, data location or register image as appropriate
   The call instruction can have several formats, which we order here
    by instruction length.  By far the most common is the direct near
    call, E8 xx xx xx xx.  Very few of these are emitted by the
    compiler.  We do not attempt to identify intersegment calls or
    calls with instruction prefix overrides.
   To accommodate other patterns we have seen in this application, we
    chase a few instructions after the call, when necessary, but we
    limit the instructions we will decoded along the way.  As new
    cases arise, reported to the log, we may expand this code.
   FF 10                CALL DWORD PTR [EAX]
   FF 11                CALL DWORD PTR [ECX]
   FF 12                CALL DWORD PTR [EDX]
   FF 13                CALL DWORD PTR [EBX]
   FF 16                CALL DWORD PTR [ESI]
   FF 17                CALL DWORD PTR [EDI]
   FF D0                CALL EAX
   FF D1                CALL ECX
   FF D2                CALL EDX
   FF D3                CALL EBX
   FF D4                CALL ESP
   FF D5                CALL EBP
   FF D6                CALL ESI
   FF D7                CALL EDI
   FF 14 24             CALL DWORD PTR [ESP]
   FF 55 xx             CALL DWORD PTR [EBP]
   FF 50 xx             CALL DWORD PTR [EAX+short]
   FF 51 xx             CALL DWORD PTR [ECX+short]
   FF 52 xx             CALL DWORD PTR [EDX+short]
   FF 53 xx             CALL DWORD PTR [EBX+short]
   FF 55 xx             CALL DWORD PTR [EBP+short]  Third most common,
```

```
tie
   FF 56 xx              CALL DWORD PTR [ESI+short]
   FF 57 xx              CALL DWORD PTR [EDI+short]
   FF 54 24 xx           CALL DWORD PTR [ESP+short]
   E8 xx xx xx xx        CALL EIP-relative           Most common
   FF 15 xx xx xx xx     CALL DWORD PTR [Address]    Second most common
   FF 90 xx xx xx xx     CALL DWORD PTR [EAX+long]
   FF 91 xx xx xx xx     CALL DWORD PTR [ECX+long]
   FF 92 xx xx xx xx     CALL DWORD PTR [EDX+long]
   FF 93 xx xx xx xx     CALL DWORD PTR [EBX+long]
   FF 95 xx xx xx xx     CALL DWORD PTR [EBP+long]   Third most common,
tie
   FF 96 xx xx xx xx     CALL DWORD PTR [ESI+long]
   FF 97 xx xx xx xx     CALL DWORD PTR [EDI+long]
   FF 94 24 xx xx xx xx  CALL DWORD PTR [esp+long]
   Jumps
   FF 20                 JMP DWORD PTR [EAX]
   FF 21                 JMP DWORD PTR [ECX]
   FF 22                 JMP DWORD PTR [EDX]
   FF 23                 JMP DWORD PTR [EBX]
   FF 26                 JMP DWORD PTR [ESI]
   FF 27                 JMP DWORD PTR [EDI]
   FF E0                 JMP EAX
   FF E1                 JMP ECX
   FF E2                 JMP EDX
   FF E3                 JMP EBX
   FF E4                 JMP ESP
   FF E5                 JMP EBP
   FF E6                 JMP ESI
   FF E7                 JMP EDI
   FF 24 24              JMP DWORD PTR [ESP]
   FF 65 xx              JMP DWORD PTR [EBP]
   FF 60 xx              JMP DWORD PTR [EAX+short]
   FF 61 xx              JMP DWORD PTR [ECX+short]
   FF 62 xx              JMP DWORD PTR [EDX+short]
   FF 63 xx              JMP DWORD PTR [EBX+short]
   FF 65 xx              JMP DWORD PTR [EBP+short]
   FF 66 xx              JMP DWORD PTR [ESI+short]
   FF 67 xx              JMP DWORD PTR [EDI+short]
   FF 64 24 xx           JMP DWORD PTR [ESP+short]
   E9 xx xx xx xx        JMP EIP-relative           Most common
   FF 25 xx xx xx xx     JMP DWORD PTR [Address]    Second most common
   FF A0 xx xx xx xx     JMP DWORD PTR [EAX+long]
   FF A1 xx xx xx xx     JMP DWORD PTR [ECX+long]
   FF A2 xx xx xx xx     JMP DWORD PTR [EDX+long]
   FF A3 xx xx xx xx     JMP DWORD PTR [EBX+long]
   FF A5 xx xx xx xx     JMP DWORD PTR [EBP+long]
   FF A6 xx xx xx xx     JMP DWORD PTR [ESI+long]
   FF A7 xx xx xx xx     JMP DWORD PTR [EDI+long]
   FF A4 24 xx xx xx xx  JMP DWORD PTR [esp+long]
A short displacement is -128 through 127, encoded in twos complement
form.  A long displacement is a twos complement 32-bit number.
*/
// Pre-compiled header files, must come first
#include "VSInit_pch.h"
#pragma hdrstop
// Compiler header files
#include <stdio.h>
// Application header files
#include "VSInit_int.h"
// Data types
typedef enum {
  INST_NONE              = 0
, CALL2_INDIR_EAX_ZERO   = 201
, CALL2_INDIR_ECX_ZERO
, CALL2_INDIR_EDX_ZERO
, CALL2_INDIR_EBX_ZERO
, CALL2_INDIR_ESI_ZERO
```

```
, CALL2_INDIR_EDI_ZERO
, CALL2_DIR_EAX
, CALL2_DIR_ECX
, CALL2_DIR_EDX
, CALL2_DIR_EBX
, CALL2_DIR_ESP
, CALL2_DIR_EBP
, CALL2_DIR_ESI
, CALL2_DIR_EDI
, CALL3_ESP_ZERO         = 301
, CALL3_EAX_SHORT
, CALL3_ECX_SHORT
, CALL3_EDX_SHORT
, CALL3_EBX_SHORT
, CALL3_EBP_SHORT
, CALL3_ESI_SHORT
, CALL3_EDI_SHORT
, CALL4_ESP_SHORT        = 401
, CALL5_EIP_RELATIVE     = 501
, JUMP5_EIP_RELATIVE
, CALL6_DGROUP           = 601
, CALL6_EAX_LONG
, CALL6_ECX_LONG
, CALL6_EDX_LONG
, CALL6_EBX_LONG
, CALL6_EBP_LONG
, CALL6_ESI_LONG
, CALL6_EDI_LONG
, JUMP6_DGROUP
, CALL7_ESP_LONG         = 701
} INST_TYPE ;
typedef struct {
  INST_TYPE it ;                              // set only for special inst
ructions
  DWORD dwAddrInst ;                          // address of current instru
ction
  DWORD dwAddrPtch ;                          // address of where to patch
} PTCH_DATA, * PPTCH_DATA ;
// Constants
const int nMaxDecodes = 4 ;                   // maximum instructions to d
ecode
// Local functions
static inline DWORD ComputeLong( DWORD dwBase, LONG lDisplacement) ;
static inline DWORD ComputeShort( DWORD dwBase, char cDisplacement) ;
static bool FollowTheCall( DWORD dwAddrStub
                         , PPTCH_DATA ppd
                         , int nStepsRemaining
                         ) ;
static bool IsCaller2( DWORD, PCPUREGS, PPTCH_DATA ppd) ;
static bool IsCaller3( DWORD, PCPUREGS, PPTCH_DATA ppd) ;
static bool IsCaller4( DWORD, PCPUREGS, PPTCH_DATA ppd) ;
static bool IsCaller5( DWORD, PCPUREGS, PPTCH_DATA ppd) ;
static bool IsCaller6( DWORD, PCPUREGS, PPTCH_DATA ppd) ;
static bool IsCaller7( DWORD, PCPUREGS, PPTCH_DATA ppd) ;
static void WarnNoPatch( int nWarning, DWORD dwAddrRet, int nbrBytes) ;
/* function BackPatch:
   Patch the caller of a static link stub
   Since each patch we apply is a single DWORD, we don't have to
    worry about yielding to another thread while a patch is half
    applied.  Is this true even if the DWORD target straddles a
    page boundary?  As of March 2003 we are called under the
    protection of a per-process critical section.
   False positives on our code matches are possible, but extremely
    unlikely.
   Returns:  true if we patched a code or data address that is likely
              to stay patched
             false if we were unable to patch, or if our patch is likely
              to fall off
```

```
             We are still fine tuning these return codes.  The caller
              uses the return code to avoid the overhead of repeated
              calls to this function for the same calling address.
*/
bool BackPatch( DWORD dwAddrRet
              , DWORD dwAddrTargetNew
              , DWORD dwAddrStub
              , PCPUREGS pRegs
              )
{
  // adjust the ESP image
  // The value pushed in our funnel is 16 bytes less than the value
  //  before the call to the stub.  The four intervening pushes are
  //  for the CALL instruction itself, the PUSH of the hash in the
  //  stub, the push of the hint in the stub, and the CALL of the stub
  //  to the funnel.  See SecurePE.asm for the latter two instructions.
  // We don't need to restore this ESP image in the stack, since the
  //  value is discarded by the caller's POPAD.  We do need to be sure
  //  we adjust the value only once, so that our calculations below are
  //  correct.
  pRegs->ESP += 16 ;
  // look for call instructions of all different lengths
  // We check all possibilities in case of ambiguity.
  PTCH_DATA pd2 = { INST_NONE, dwAddrRet - 2 } ;
  PTCH_DATA pd3 = { INST_NONE, dwAddrRet - 3 } ;
  PTCH_DATA pd4 = { INST_NONE, dwAddrRet - 4 } ;
  PTCH_DATA pd5 = { INST_NONE, dwAddrRet - 5 } ;
  PTCH_DATA pd6 = { INST_NONE, dwAddrRet - 6 } ;
  PTCH_DATA pd7 = { INST_NONE, dwAddrRet - 7 } ;
  bool fFound2 = IsCaller2( dwAddrStub, pRegs, &pd2) ;
  bool fFound3 = IsCaller3( dwAddrStub, pRegs, &pd3) ;
  bool fFound4 = IsCaller4( dwAddrStub, pRegs, &pd4) ;
  bool fFound5 = IsCaller5( dwAddrStub, pRegs, &pd5) ;
  bool fFound6 = IsCaller6( dwAddrStub, pRegs, &pd6) ;
  bool fFound7 = IsCaller7( dwAddrStub, pRegs, &pd7) ;
  // ensure there is exactly one match
  int nbrMatches = 0
    , biggestMatch = 0
    ;
  PPTCH_DATA ppd ;
  if ( fFound2)  { ppd = &pd2 ; nbrMatches++ ; biggestMatch = 2 ; }
  if ( fFound3)  { ppd = &pd3 ; nbrMatches++ ; biggestMatch = 3 ; }
  if ( fFound4)  { ppd = &pd4 ; nbrMatches++ ; biggestMatch = 4 ; }
  if ( fFound5)  { ppd = &pd5 ; nbrMatches++ ; biggestMatch = 5 ; }
  if ( fFound6)  { ppd = &pd6 ; nbrMatches++ ; biggestMatch = 6 ; }
  if ( fFound7)  { ppd = &pd7 ; nbrMatches++ ; biggestMatch = 7 ; }
  if ( nbrMatches == 0)  {
    WarnNoPatch( WARNING_SECURE_BACK_PATCH_1, dwAddrRet, 7) ;
    return false ;
  }
  if ( nbrMatches > 1)  {
    WarnNoPatch( WARNING_SECURE_BACK_PATCH_2, dwAddrRet, biggestMatch) ;
    return false ;
  }
  // handle the exception cases
  // These are direct calls or jumps through registers, and EIP-
  //  relative calls and jumps.
  // now apply the patch, all too many cases
  // Only the special cases set ppd->it.
  //
  DWORD dwAddrPatch = ppd->dwAddrPtch ;     // where to patch, really PD
WORD
  DWORD dwNewValue = dwAddrTargetNew ;      // for all cases but most co
mmon
  PBYTE pbInst = PBYTE( dwAddrRet - biggestMatch) ;
  switch ( ppd->it)  {                      // switch on the instruction
 type
    default :  break ;
```

```
    case CALL2_DIR_EAX : pRegs->EAX = dwNewValue ; return false ;
    case CALL2_DIR_ECX : pRegs->ECX = dwNewValue ; return false ;
    case CALL2_DIR_EDX : pRegs->EDX = dwNewValue ; return false ;
    case CALL2_DIR_EBX : pRegs->EBX = dwNewValue ; return false ;
    case CALL2_DIR_EBP : pRegs->EBP = dwNewValue ; return false ;
    case CALL2_DIR_ESI : pRegs->ESI = dwNewValue ; return false ;
    case CALL2_DIR_EDI : pRegs->EDI = dwNewValue ; return false ;
    case CALL2_DIR_ESP :
      WarnNoPatch( WARNING_SECURE_BACK_PATCH_4, dwAddrRet, 2) ;
      return false ;                          // cannot easily change ESP
    case JUMP5_EIP_RELATIVE :
    case CALL5_EIP_RELATIVE :
      dwNewValue = dwAddrTargetNew - ( dwAddrPatch + 4) ;
      break ;
  }  // switch on the call type
  // apply the patch
  if ( PatchDWord( dwAddrPatch, dwNewValue) == false)  {
    WarnNoPatch( WARNING_SECURE_BACK_PATCH_3, dwAddrRet, biggestMatch) ;
    return false ;                            // return to caller
  }
#if 0                                         // testing only
  // see which patches we did apply
  LogSecError( "%u %X %X"                     // this will flood the log
             , INFO_SECURE_GOOD_BACK_PATCH
             , dwAddrPatch
             , dwNewValue
             ) ;
#endif
  // successful return
  return true ;
}  // BackPatch
/* function PatchDWord:
   Write a DWORD to an address that may be on a read-only page
*/
bool PatchDWord( DWORD dwAddrTarget, DWORD dwValue)
{
  // write to memory
  // We don't need to lock the patched page(s), even if it is code.
  //  The page is marked dirty by our modification, and this appears
  //  to be enough to make the memory manager assign a backing page
  //  in the swap file if one is not already assigned.
  //  In NT/XP/2K, the write also triggers copy-on-write processing
  //   for code pages.
  PDWORD pdwTarget = PDWORD( dwAddrTarget) ;
  DWORD flOldProtect ;
  if ( VirtualProtect( LPVOID( pdwTarget)
                     , sizeof( DWORD)
                     , PAGE_EXECUTE_READWRITE
                     , &flOldProtect
                     ) == FALSE)  {
    return false ;
  }  // if VirtualProtect failed
  *pdwTarget = dwValue ;
  if ( VirtualProtect( LPVOID( pdwTarget)
                     , sizeof( DWORD)
                     , flOldProtect
                     , &flOldProtect
                     ) == FALSE)  {
    return false ;
  }
  // successful return
  return true ;
}  // PatchDWord
// All functions below are private to this file.
/* function ComputeLong:
   Compute an address from a base address and a signed 32-bit displaceme
nt
   Because of 32-bit wraparound, we can just as soon treat the displacem
```

```
ent
    as unsigned.
*/
static inline DWORD ComputeLong( DWORD dwBase, LONG lDisplacement)
{
  return dwBase + lDisplacement ;
}  // ComputeLong
/* function ComputeShort:
   Compute an address from a base address and a signed character displac
ement
*/
static inline DWORD ComputeShort( DWORD dwBase, char cDisplacement)
{
  return dwBase + LONG( cDisplacement) ;
}  // ComputeShort
/* function FollowTheCall:
   Simulate a few instructions to see if the call reaches its target
   Since this function is called inside a __try / __except scope, we
    reference memory with impunity.
*/
static bool FollowTheCall( DWORD dwAddrStub
                         , PPTCH_DATA ppd
                         , int nStepsRemaining
                         )
{
  // bail out if we have decoded enough already
  // This avoids excessive path length in general, and loops in
  //  particular.
  if ( nStepsRemaining <= 0)
    return false ;
  // point to the instruction
  PBYTE pbInst = PBYTE( ppd->dwAddrInst) ;
  // indirect near jump?
  if ( pbInst[ 0] == 0xff  &&  pbInst[ 1] == 0x25)  {
    // update the patch address if a prior instruction cannot be skipped
    if ( ppd->dwAddrPtch == 0)
      ppd->dwAddrPtch = * PDWORD( ppd->dwAddrInst + 2) ;
    // if this hits the target
    DWORD dwTarget = * PDWORD( * PDWORD( pbInst + 2)) ;
    if ( dwTarget == dwAddrStub)
      return true ;
    // try the next instruction
    ppd->dwAddrInst = dwTarget ;             // simulate the instruction
    return FollowTheCall( dwAddrStub, ppd, nStepsRemaining - 1) ;
  }  // indirect near jump
  // direct near jump?
  if ( pbInst[ 0] == 0xe9)  {
    // update the patch address if a prior instruction cannot be skipped
    if ( ppd->dwAddrPtch == 0)  {
      ppd->dwAddrPtch = ppd->dwAddrInst + 1 ;// was 2, fixes bug 18929
      ppd->it = JUMP5_EIP_RELATIVE ;         // gets special handling
    }
    // if this hits the target
    DWORD dwTarget = ppd->dwAddrInst + 5 + * ( PDWORD)( pbInst + 1) ;
    if ( dwTarget == dwAddrStub)
      return true ;
    // try the next instruction
    ppd->dwAddrInst = dwTarget ;             // simulate the instruction
    return FollowTheCall( dwAddrStub, ppd, nStepsRemaining - 1) ;
  }  // indirect near jump
  // imm-32 register load?
  if ( ( pbInst[ 0] & 0xf8) == 0xb8)  {
    // bail out if ESP is the destination
    if ( pbInst[ 0] == 0xbc)
      return false ;
    // advance the instruction pointer
    ppd->dwAddrInst += 5 ;
    // erase memory of the call instruction, since any patch must be
```

```
    //  applied to a subsequent jump, lest the patch skip this
    //  instruction
    ppd->it = INST_NONE ;
    ppd->dwAddrPtch = 0 ;
    // try the next instruction
    return FollowTheCall( dwAddrStub, ppd, nStepsRemaining - 1) ;
  }  // imm-32 register load
  // some imm-32 moves to memory
  if ( pbInst[ 0] == 0xc7)  {
    DWORD dwInstBytes ;                     // bytes in the instruction
    switch ( pbInst[ 1])  {
      default   : return false ;            // we don't handle yet
      case 0x00 :                           // [EAX]
      case 0x01 :                           // [ECX]
      case 0x02 :                           // [EDX]
      case 0x03 :                           // [EBX]
      case 0x06 :                           // [ESI]
      case 0x07 :                           // [EDI]
        dwInstBytes = 6 ;
        break ;
      case 0x40 :                           // [EAX+short]
      case 0x41 :                           // [ECX+short]
      case 0x42 :                           // [EDX+short]
      case 0x43 :                           // [EBX+short]
      case 0x45 :                           // [EBP+short] or [EBP]
      case 0x46 :                           // [ESI+short]
      case 0x47 :                           // [EDI+short]
        dwInstBytes = 7 ;
        break ;
      case 0x80 :                           // [EAX+long]
      case 0x81 :                           // [ECX+long]
      case 0x82 :                           // [EDX+long]
      case 0x83 :                           // [EBX+long]
      case 0x85 :                           // [EBP+long]
      case 0x86 :                           // [ESI+long]
      case 0x87 :                           // [EDI+long]
        dwInstBytes = 10 ;
        break ;
    }  // switch on the instruction type
    // advance the instruction pointer
    ppd->dwAddrInst += dwInstBytes ;
    // erase memory of the call instruction, since any patch must be
    //  applied to a subsequent jump, lest the patch skip this
    //  instruction
    ppd->it = INST_NONE ;
    ppd->dwAddrPtch = 0 ;
    // try the next instruction
    return FollowTheCall( dwAddrStub, ppd, nStepsRemaining - 1) ;
  }  // some imm-32 moves to memory
  // we did not find an instruction of interest
  return false ;
}  // FollowTheCall
/* function IsCaller2:
   Could the caller have used a 2-byte instruction?
   When one function calls another several times in succession, the
    compiler can generate small code by putting the target address
    in a register and calling repeatedly through that register.  In
    VC++ 6.0 we have seen this technique used only for imported
    functions, which our stubs are not.
*/
static bool IsCaller2( DWORD dwAddrStub, PCPUREGS pRegs, PPTCH_DATA ppd)
{
  // ensure we can read the instruction bytes
  PBYTE pbInst = PBYTE( ppd->dwAddrInst) ;
  if ( IsBadReadPtr( pbInst, 2))
    return false ;
  // if there is a 2-byte call here, decide which one
  if ( pbInst[ 0] != 0xff)
```

```
    return false ;
  DWORD dwReg ;
  switch ( pbInst[ 1])  {
    default   : return false ;
    case 0x10 : dwReg = pRegs->EAX ; break ;
    case 0x11 : dwReg = pRegs->ECX ; break ;
    case 0x12 : dwReg = pRegs->EDX ; break ;
    case 0x13 : dwReg = pRegs->EBX ; break ;
    case 0x16 : dwReg = pRegs->ESI ; break ;
    case 0x17 : dwReg = pRegs->EDI ; break ;
    case 0xD0 : ppd->it = CALL2_DIR_EAX ; dwReg = pRegs->EAX ; break ;
    case 0xD1 : ppd->it = CALL2_DIR_ECX ; dwReg = pRegs->ECX ; break ;
    case 0xD2 : ppd->it = CALL2_DIR_EDX ; dwReg = pRegs->EDX ; break ;
    case 0xD3 : ppd->it = CALL2_DIR_EBX ; dwReg = pRegs->EBX ; break ;
    case 0xD4 : ppd->it = CALL2_DIR_ESP ; dwReg = pRegs->ESP ; break ;
    case 0xD5 : ppd->it = CALL2_DIR_EBP ; dwReg = pRegs->EBP ; break ;
    case 0xD6 : ppd->it = CALL2_DIR_ESI ; dwReg = pRegs->ESI ; break ;
    case 0xD7 : ppd->it = CALL2_DIR_EDI ; dwReg = pRegs->EDI ; break ;
  }  // switch on the second byte
  // validate the possible instruction's target
  if ( ( pbInst[ 1] & 0x80) != 0)  {        // if direct call through re
gister
    // we are done if this hits the target
    if ( dwReg == dwAddrStub)               // if the call is to the stu
b
      return true ;
    // try following the call
    ppd->dwAddrInst = dwReg ;               // simulate the call
    return FollowTheCall( dwAddrStub, ppd, nMaxDecodes) ;
  }
  // set the patch address
  ppd->dwAddrPtch = dwReg ;
  // possible instruction is an indirect call
  __try  {
    // we are done if this hits the target
    if ( * PDWORD( dwReg) == dwAddrStub)    // if the call is to the stu
b
      return true ;
    // try following the call
    ppd->dwAddrInst = * PDWORD( dwReg) ;    // simulate the call
    return FollowTheCall( dwAddrStub, ppd, nMaxDecodes) ;
  }  // __try
  __except ( EXCEPTION_EXECUTE_HANDLER)  {
    return false ;                          // if no instruction validat
ed
  }  // __except
}  // IsCaller2
/* function IsCaller3:
   Could the caller have used a 3-byte instruction?
*/
static bool IsCaller3( DWORD dwAddrStub, PCPUREGS pRegs, PPTCH_DATA ppd)
{
  // ensure we can read the instruction bytes
  PBYTE pbInst = PBYTE( ppd->dwAddrInst) ;
  if ( IsBadReadPtr( pbInst, 3))
    return false ;
  // if there is a 3-byte call here, decide which one
  if ( pbInst[ 0] != 0xff)
    return false ;
  DWORD dwReg ;
  switch ( pbInst[ 1])  {
    default   : return false ;
    case 0x14 : dwReg = pRegs->ESP ; break ;
    case 0x50 : dwReg = pRegs->EAX ; break ;
    case 0x51 : dwReg = pRegs->ECX ; break ;
    case 0x52 : dwReg = pRegs->EDX ; break ;
    case 0x53 : dwReg = pRegs->EBX ; break ;
    case 0x55 : dwReg = pRegs->EBP ; break ;
```

```
    case 0x56 : dwReg = pRegs->ESI ; break ;
    case 0x57 : dwReg = pRegs->EDI ; break ;
  }  // switch on the second byte
  // determine the patch address
  if ( pbInst[ 1] == 0x14)
    ppd->dwAddrPtch = dwReg ;                // CALL3_ESP_ZERO
  else
    ppd->dwAddrPtch = ComputeShort( dwReg, pbInst[ 2]) ;
  // validate the possible instruction's target
  LONG lDisplacement ;
  if ( pbInst[ 1] == 0x14)                   // CALL3_ESP_ZERO
    lDisplacement = 0 ;
  else
    lDisplacement = LONG( pbInst[ 2]) ;
  // possible instruction is an indirect call
  __try  {
    DWORD dwTarget = * PDWORD( dwReg + lDisplacement) ;
    if ( dwTarget == dwAddrStub)             // if the call is to the stu
b
      return true ;
    // try following the call
    ppd->dwAddrInst = dwTarget ;             // simulate the call
    return FollowTheCall( dwAddrStub, ppd, nMaxDecodes) ;
  }  // __try
  __except ( EXCEPTION_EXECUTE_HANDLER)  {
    return false ;                           // if no instruction validat
ed
  }  // __except
}  // IsCaller3
/* function IsCaller4:
   Could the caller have used a 4-byte instruction?
*/
static bool IsCaller4( DWORD dwAddrStub, PCPUREGS pRegs, PPTCH_DATA ppd)
{
  // ensure we can read the instruction bytes
  PBYTE pbInst = PBYTE( ppd->dwAddrInst) ;
  if ( IsBadReadPtr( pbInst, 4))
    return false ;
  // there is only one 4-byte call
  if ( pbInst[ 0] != 0xff  ||  pbInst[ 1] != 0x54  ||  pbInst[ 2] != 0x2
4)
    return false ;
  DWORD dwReg = pRegs->ESP ;
  ppd->dwAddrPtch = ComputeShort( dwReg, pbInst[ 3]) ;
  LONG lDisplacement = LONG( pbInst[ 3]) ;
  __try  {
    DWORD dwTarget = * PDWORD( dwReg + lDisplacement) ;
    if ( dwTarget == dwAddrStub)             // if the call is to the stu
b
      return true ;
    // try following the call
    ppd->dwAddrInst = dwTarget ;             // simulate the call
    return FollowTheCall( dwAddrStub, ppd, nMaxDecodes) ;
  }  // __try
  __except ( EXCEPTION_EXECUTE_HANDLER)  {
    return false ;                           // if no instruction validat
ed
  }  // __except
}  // IsCaller4
/* function IsCaller5:
   Could the caller have used a 5-byte instruction?
*/
static bool IsCaller5( DWORD dwAddrStub, PCPUREGS pRegs, PPTCH_DATA ppd)
{
  // ensure we can read the instruction bytes
  PBYTE pbInst = PBYTE( ppd->dwAddrInst) ;
  if ( IsBadReadPtr( pbInst, 5))
    return false ;
```

```
  // is there is a 5-byte call here?
  if ( pbInst[ 0] != 0xe8)
    return false ;
  // set the patch target and instruction
  // These will be overridden only if we step into instructions that
  //  cannot be skipped.
  ppd->dwAddrPtch = ppd->dwAddrInst + 1 ;
  ppd->it = CALL5_EIP_RELATIVE ;            // gets special handling
  // see if this call goes directly to the stub
  DWORD dwTarget = ppd->dwAddrInst + 5 + * PDWORD( pbInst + 1) ;
  if ( dwTarget == dwAddrStub)              // if the call is to the stu
b
    return true ;
  // try following the call
  __try  {                                  // __try / __except
    ppd->dwAddrInst = dwTarget ;            // simulate the call
    return FollowTheCall( dwAddrStub, ppd, nMaxDecodes) ;
  }  // __try
  __except ( EXCEPTION_EXECUTE_HANDLER)  {
    return false ;
  }  // __except
}  // IsCaller5
/* function IsCaller6:
   Could the caller have used a 6-byte instruction?
*/
static bool IsCaller6( DWORD dwAddrStub, PCPUREGS pRegs, PPTCH_DATA ppd)
{
  // ensure we can read the instruction bytes
  PBYTE pbInst = PBYTE( ppd->dwAddrInst) ;
  if ( IsBadReadPtr( pbInst, 6))
    return false ;
  // if there is a 6-byte call here, decide which one
  if ( pbInst[ 0] != 0xff)
    return false ;
  DWORD dwReg ;
  switch ( pbInst[ 1])  {
    default   : return false ;
    case 0x15 : dwReg = 0          ; break ;
    case 0x90 : dwReg = pRegs->EAX ; break ;
    case 0x91 : dwReg = pRegs->ECX ; break ;
    case 0x92 : dwReg = pRegs->EDX ; break ;
    case 0x93 : dwReg = pRegs->EBX ; break ;
    case 0x95 : dwReg = pRegs->EBP ; break ;
    case 0x96 : dwReg = pRegs->ESI ; break ;
    case 0x97 : dwReg = pRegs->EDI ; break ;
  }  // switch on the second byte
  // determine the patch address
  ppd->dwAddrPtch = ComputeLong( dwReg, * PDWORD( pbInst + 2)) ;
  // validate the possible instruction's target
  // For the DGROUP call, the vector must be in the DGROUP.
  PDWORD pdwTarget = PDWORD( ppd->dwAddrPtch) ;
#if 0                                       // add this test
  if ( pbInst[ 1] == 0x15)  {
    __try  {
      IsAddressInThisModule( DWORD( pdwTarget)) ;
    }  // __try
    __except ( EXCEPTION_EXECUTE_HANDLER)  {
      return false ;
    }  // __except
  }  // if a possible DGROUP call
#endif
  // validate the target for all instructions
  __try  {
    DWORD dwTarget = *pdwTarget ;
    if ( dwTarget == dwAddrStub)            // if the call is to the stu
b
      return true ;
    // try following the call
```

```
    ppd->dwAddrInst = dwTarget ;             // simulate the instruction
    return FollowTheCall( dwAddrStub, ppd, nMaxDecodes) ;
  }  // __try
  __except ( EXCEPTION_EXECUTE_HANDLER)  {
    return false ;
  }  // __except
}  // IsCaller6
/* function IsCaller7:
   Could the caller have used a 7-byte instruction?
*/
static bool IsCaller7( DWORD dwAddrStub, PCPUREGS pRegs, PPTCH_DATA ppd)
{
  // ensure we can read the instruction bytes
  PBYTE pbInst = PBYTE( ppd->dwAddrInst) ;
  if ( IsBadReadPtr( pbInst, 7))
    return false ;
  // there is only one 7-byte call
  if ( pbInst[ 0] != 0xff  ||  pbInst[ 1] != 0x94  ||  pbInst[ 2] != 0x2
4)
    return false ;
  // compute the patch address
  ppd->dwAddrPtch = ComputeLong( pRegs->ESP, * PDWORD( pbInst + 3)) ;
  // validate the possible instruction's target
  __try  {
    DWORD dwTarget = * PDWORD( ppd->dwAddrPtch) ;
    if ( dwTarget == dwAddrStub)             // if the call is to the stu
b
      return true ;
    // try following the call
    ppd->dwAddrInst = dwTarget ;             // simulate the call
    return FollowTheCall( dwAddrStub, ppd, nMaxDecodes) ;
  }  // __try
  __except ( EXCEPTION_EXECUTE_HANDLER)  {
    return false ;
  }  // __except
}  // IsCaller7
/* function WarnNoPatch:
   Warn that we were unable to back patch the static link caller
*/
static void WarnNoPatch( int nWarning, DWORD dwAddrRet, int nbrBytes)
{
  // point to the earliest possible instruction start
  PBYTE pbInst = PBYTE( dwAddrRet - nbrBytes) ;
  // format the first part of the message
  char szMsg[ 100] ;
  int iLength = 0 ;
  // display the instruction bytes if possible
  __try  {
    for ( int ndxByte = 0 ; ndxByte < nbrBytes ; ndxByte++)  {
      iLength += sprintf( szMsg + iLength, "%2.2X ", pbInst[ ndxByte]) ;
    }
  }  // __try
  __except ( EXCEPTION_EXECUTE_HANDLER)  {
  }  // __except
  // report the error
  LogSecError( "%u %X %s", nWarning, pbInst, szMsg) ;
}  // WarnNoPatch
// Certificates.cpp
// Copyright (c) 2004. Zone Labs, LLC All Rights Reserved.
/* File Certificates.cpp:
   Manage certificates for module validation
*/
// Pre-compiled header files, must come first
#include "VSInit_pch.h"
#pragma hdrstop
// Windows header files
#include <wincrypt.h>
#include <wintrust.h>
```

```
// Compiler header files
// Application header files
#include <vsinit.h>
#include "vsinit_int.h"
// Constants
// The compiler emits bloated code for aggregate constants (e.g., arrays
//  or structures) in function scope.
const char szKeyTest[] = "Software\\Microsoft\\Windows\\CurrentVersion"
                         "\\WinTrust\\Trust Providers\\Software Publishi
ng"
         , szValState[] = "State"
         , szSysStoreRoot[] = "Root"
         , szSysStoreTrust[] = "Trust"
         , szSysStoreCA[] = "CA"
         , szMSVCRT[] = "MSVCRT"            // C RTL DLL, release build
         , szMSVCRTD[] = "MSVCRTD"          // C RTL DLL, debug build
         ;
const char * pszSysStore[] = { szSysStoreRoot
                             , szSysStoreTrust
                             , szSysStoreCA
                             } ;
const int nbrSysStores = sizeof pszSysStore / sizeof pszSysStore[ 0] ;
// If we define these arrays as const, the compiler complains because
//  that disagrees with the CRYPT_ALGORITHM_IDENTIFIER and CRYPT_BIT_
//  BLOB definitions.
static char szPubKeyTestAlgID[] = szOID_RSA_RSA ;
static BYTE bPubKeyTest1[]                  // public key for test certi
ficate
           = { 0x30, 0x47, 0x02, 0x40, 0x81, 0x55, 0x22, 0xB9, 0x8A, 0x
A4
             , 0x6F, 0xED, 0xD6, 0xE7, 0xD9, 0x66, 0x0F, 0x55, 0xBC, 0x
D7
             , 0xCD, 0xD5, 0xBC, 0x4E, 0x40, 0x02, 0x21, 0xA2, 0xB1, 0x
F7
             , 0x87, 0x30, 0x85, 0x5E, 0xD2, 0xF2, 0x44, 0xB9, 0xDC, 0x
9B
             , 0x75, 0xB6, 0xFB, 0x46, 0x5F, 0x42, 0xB6, 0x9D, 0x23, 0x
36
             , 0x0B, 0xDE, 0x54, 0x0F, 0xCD, 0xBD, 0x1F, 0x99, 0x2A, 0x
10
             , 0x58, 0x11, 0xCB, 0x40, 0xCB, 0xB5, 0xA7, 0x41, 0x02, 0x
03
             , 0x01, 0x00, 0x01
             }
         , bPubKeyTest2[]                   // public key for test certi
ficate
           = { 0x30, 0x48, 0x02, 0x41, 0x00, 0x81, 0x55, 0x22, 0xB9, 0x
8A
             , 0xA4, 0x6F, 0xED, 0xD6, 0xE7, 0xD9, 0x66, 0x0F, 0x55, 0x
BC
             , 0xD7, 0xCD, 0xD5, 0xBC, 0x4E, 0x40, 0x02, 0x21, 0xA2, 0x
B1
             , 0xF7, 0x87, 0x30, 0x85, 0x5E, 0xD2, 0xF2, 0x44, 0xB9, 0x
DC
             , 0x9B, 0x75, 0xB6, 0xFB, 0x46, 0x5F, 0x42, 0xB6, 0x9D, 0x
23
             , 0x36, 0x0B, 0xDE, 0x54, 0x0F, 0xCD, 0xBD, 0x1F, 0x99, 0x
2A
             , 0x10, 0x58, 0x11, 0xCB, 0x40, 0xCB, 0xB5, 0xA7, 0x41, 0x
02
             , 0x03, 0x01, 0x00, 0x01
             }
         , bPubKeyTest3[]                   // public key for test certi
ficate
           = { 0x30, 0x47, 0x02, 0x40, 0x9C, 0x50, 0x05, 0x1D, 0xE2, 0x
0E
             , 0x4C, 0x53, 0xD8, 0xD9, 0xB5, 0xE5, 0xFD, 0xE9, 0xE3, 0x
AD
```

```
              , 0x83, 0x4B, 0x80, 0x08, 0xD9, 0xDC, 0xE8, 0xE8, 0x35, 0x
F8
              , 0x11, 0xF1, 0xE9, 0x9B, 0x03, 0x7A, 0x65, 0x64, 0x76, 0x
35
              , 0xCE, 0x38, 0x2C, 0xF2, 0xB6, 0x71, 0x9E, 0x06, 0xD9, 0x
BF
              , 0xBB, 0x31, 0x69, 0xA3, 0xF6, 0x30, 0xA0, 0x78, 0x7B, 0x
18
              , 0xDD, 0x50, 0x4D, 0x79, 0x1E, 0xEB, 0x61, 0xC1, 0x02, 0x
03
              , 0x01, 0x00, 0x01
              }
          ;
const CERT_PUBLIC_KEY_INFO CertTestKey1
      = { { szPubKeyTestAlgID, 0}, { sizeof bPubKeyTest1, bPubKeyTest1,
0 }} ;
const CERT_PUBLIC_KEY_INFO CertTestKey2
      = { { szPubKeyTestAlgID, 0}, { sizeof bPubKeyTest2, bPubKeyTest2,
0 }} ;
const CERT_PUBLIC_KEY_INFO CertTestKey3
      = { { szPubKeyTestAlgID, 0}, { sizeof bPubKeyTest3, bPubKeyTest3,
0 }} ;
// Global data
HMODULE hmodCrypt32 = 0 ;                    // does not exist in base OS
R2
// Local data
static bool fTestCertAllowed = false ;       // true if test certificates
 allowed
static HCERTSTORE hSysStore[ nbrSysStores + 1] = { 0} ;// null terminato
r
// Local functions
static bool CloseCertificateStores() ;
static __declspec( noreturn) void ErrorLoad( const char * pszModName) ;
static bool IsATestCert( PCERT_PUBLIC_KEY_INFO pCertPubKeyInfo) ;
static bool IsCertValidInThisStore( HCERTSTORE hCertStore
                                  , PBYTE pbCertData
                                  , DWORD cbCertData
                                  ) ;
static bool IsTestCertAllowed() ;
static bool LoadCrypt32() ;
static bool OpenCertificateStores() ;
static void __cdecl TerminateCertificates() ;
/* function InitializeCertificates:
   Static object initialization
   Prepare data structures for certificate management.
   This function is called from the carefully sequenced static object
    initialization in VSInit.cpp.
*/
bool InitializeCertificates()
{
  // set the function pointers
  LoadCrypt32() ;
  // open the well-known system certificate stores
  // Open:  Raise an exception if the call fails, since the call does
  //         not raise its own exceptions.
  if ( OpenCertificateStores() == false)
    LogSecFatal( "%u", ERROR_SECURE_OPEN_CERT_STORES) ;
  // remember if test certificates are allowed
  fTestCertAllowed = IsTestCertAllowed() ;
  // workaround for bug 9028
  // Increment MSVCRT/MSVCRTD reference count to keep the DLL from being
  //  unloaded too early in Win95 OSR2.  In our test, the FreeLibrary
  //  call for Crypt32.dll also freed a page containing the current
  //  module's __onexitbegin table.
  HMODULE hmodMSVCRT = GetModuleHandle( szMSVCRT) ;
  if ( hmodMSVCRT != 0)                      // if release module is pres
ent
    LoadLibrary( szMSVCRT) ;                 // bump its reference count
```

```
  else  {                                       // if MSVCRT is not loaded
    hmodMSVCRT = GetModuleHandle( szMSVCRTD) ;// try the debug module
    if ( hmodMSVCRT != 0)                       // if debug module is presen
t
      LoadLibrary( szMSVCRTD) ;                 // bump its reference count
  }  // if MSVCRT is not loaded
  // specify the termination function
  atexit( TerminateCertificates) ;
  // successful return
  return true ;
}  // InitializeCertificates
/* function IsCertValidInAnyStore:
   Validate a certificate against a list of certificate stores
   One successful validation is all we require.
*/
bool IsCertValidInAnyStore( PBYTE pbCertData
                          , DWORD dwDataLength
                          , HCERTSTORE hStoreMsg
                          )
{
  // first try the message's certificate store
  if ( IsCertValidInThisStore( hStoreMsg, pbCertData, dwDataLength))
    return true ;
  // now try the common certificate stores
  int ndxSysStore ;
  for ( ndxSysStore = 0 ; hSysStore[ ndxSysStore] != 0 ; ndxSysStore++)
 {
    if ( IsCertValidInThisStore( hSysStore[ ndxSysStore]
                               , pbCertData
                               , dwDataLength
                               ))  {
      return true ;
    }
  }  // loop once for each certificate store
  // error return
  return false ;
}  // IsCertValidInAnyStore
// All functions below are private to this file
/* function IsCertValidInThisStore:
   Validate a certificate against the specified certificate store
*/
static bool IsCertValidInThisStore( HCERTSTORE hCertStore
                                  , PBYTE pbCertData
                                  , DWORD cbCertData
                                  )
{
  PCCERT_CONTEXT pCertContext = 0
               , pCertCtxPrev = 0
               ;
  __try  {
    // create a context from this certificate
    pCertContext
      = CertCreateCertificateContext( X509_ASN_ENCODING | PKCS_7_ASN_ENC
ODING
                                    , pbCertData
                                    , cbCertData
                                    ) ;
    if ( pCertContext == 0)
      return false ;
    // look for an issuer
    // MSDN recommends the certificate chain verification functions, but
    //  but we stick with a technique that also supports Win9x with IE
    //  4.x.
    PCCERT_CONTEXT pCertCtxCurr ;
    do  {
      DWORD dwCertFlags = CERT_STORE_SIGNATURE_FLAG ;
      pCertCtxCurr =
        CertGetIssuerCertificateFromStore( hCertStore
```

```
                                        , pCertContext
                                        , pCertCtxPrev
                                        , &dwCertFlags
                                        ) ;
      pCertCtxPrev = pCertCtxCurr ;
      DWORD dwLastError = GetLastError() ;  // save a possible error cod
e
      // decide if this one passed muster
      // We also handle the special case of a self-signed certificate.
      if ( ( dwCertFlags & CERT_STORE_SIGNATURE_FLAG) == 0)  {
        if ( pCertCtxPrev != 0)  {          // if the call succeeded
          // if test certificates are not allowed, check this issuer
          if ( fTestCertAllowed)            // if allowed
            return true ;                   // no need to check
          return !IsATestCert( &pCertCtxPrev->pCertInfo->SubjectPublicKe
yInfo) ;
        }  // if issuer found and certificate is valid
        // special case for self-signed certificate
        // WinCrypt.h notes that in this case the signature is still
        //  verified.
        if ( dwLastError == CRYPT_E_SELF_SIGNED)// self-signed okay with
 us
          return true ;                     // we are done
      }  // if the certificate may have been validated
    } while ( pCertCtxPrev != 0) ;
  }  // __try
  __finally  {
    if ( pCertCtxPrev != 0)
      CertFreeCertificateContext( pCertCtxPrev) ;// free the last one
    if ( pCertContext != 0)
      CertFreeCertificateContext( pCertContext) ;
  }  // __finally
  // error return
  return false ;
}  // IsCertValidInThisStore
/* function CloseCertificateStores:
   Close the well-known system certificate stores
*/
static bool CloseCertificateStores()
{
  // open all the system stores we can
  int ndxSysStore
    , ndxHandle = 0
    ;
  for ( ndxSysStore = 0 ; ndxSysStore < nbrSysStores ; ndxSysStore++)  {
    if ( hSysStore[ ndxSysStore] == 0)      // if this store not open
      continue ;                            // go to the next
    CertCloseStore( hSysStore[ ndxSysStore], 0) ;
    hSysStore[ ndxSysStore] = 0 ;           // zero the handle
  }  // loop once for each system store
  // successful return
  return true ;
}  // CloseCertificateStores
/* function ErrorLoad:
   Report a LoadLibrary error, then raise an exception
*/
__declspec( noreturn)
void ErrorLoad( const char * pszModName)
{
  LogSecFatal( "%u: %d %s"
             , ERROR_SECURE_LOAD_LIBRARY
             , GetLastError()
             , pszModName
             ) ;
}  // ErrorLoad
/* function IsATestCert:
   Is this public key from a test certificate?
*/
```

```
static bool IsATestCert( PCERT_PUBLIC_KEY_INFO pCertPubKeyInfo)
{
  // run the gauntlet of the three public keys used in test certificates
  const DWORD dwCertEncodingType = X509_ASN_ENCODING | PKCS_7_ASN_ENCODI
NG ;
  if ( CertComparePublicKeyInfo( dwCertEncodingType
                               , pCertPubKeyInfo
                               , PCERT_PUBLIC_KEY_INFO( &CertTestKey1)
                               ))  {
    LogSecError( "%u", ERROR_SECURE_TEST_CERT_1) ;
    return true ;
  }  // if a test certificate
  if ( CertComparePublicKeyInfo( dwCertEncodingType
                               , pCertPubKeyInfo
                               , PCERT_PUBLIC_KEY_INFO( &CertTestKey2)
                               ))  {
    LogSecError( "%u", ERROR_SECURE_TEST_CERT_2) ;
    return true ;
  }  // if a test certificate
  if ( CertComparePublicKeyInfo( dwCertEncodingType
                               , pCertPubKeyInfo
                               , PCERT_PUBLIC_KEY_INFO( &CertTestKey3)
                               ))  {
    LogSecError( "%u", ERROR_SECURE_TEST_CERT_3) ;
    return true ;
  }  // if a test certificate
  // this is not a test certificate
  return false ;
}  // IsATestCert
/* function IsTestCertAllowed:
   Check the same registry flag as WinVerifyTrust does
*/
static bool IsTestCertAllowed()
{
  // open the key
  HKEY hkey ;
  if ( RegOpenKeyEx( HKEY_CURRENT_USER
                   , szKeyTest
                   , 0                        // reserved
                   , KEY_QUERY_VALUE
                   , &hkey
                   ) != ERROR_SUCCESS)  {
    return false ;
  }  // if RegOpenKeyEx failed
  // get the value
  DWORD dwType
      , dwState
      , cbData = sizeof( dwState)
      ;
  if ( RegQueryValueEx( hkey
                      , szValState
                      , 0                     // lpReserved
                      , &dwType
                      , PBYTE( &dwState)
                      , &cbData
                      ) != ERROR_SUCCESS)  {
    RegCloseKey( hkey) ;                      // close the key
    return false ;
  }  // if RegQueryValueEx failed
  // close the key
  RegCloseKey( hkey) ;
  // validate the value type and length
  if ( dwType != REG_DWORD)
    return false ;
  if ( cbData != sizeof( dwState))
    return false ;
  // return the status of the test flag
  return ( dwState & WTPF_TRUSTTEST) != 0 ;
```

```
}  // IsTestCertAllowed
/* function LoadCrypt32:
   Load Crypt32.dll if it exists
   Integrity clients must support base OSR2, which lacks Crypt32.dll.
    OSR2 first got Crypt32.dll with IE 4.0.
   We must also ensure that the DLL is not partially loaded as the
    result of a static link from another module.
*/
static bool LoadCrypt32()
{
  // common variables
  char szSysDir[ MAX_PATH + 1] ;
  char szModName[ MAX_PATH + 20] ;
  // load Crypt32.dll
  // Load from the system directory, to keep an attacker from putting
  //  bogus local copies in a private directory.
  GetSystemDirectory( szSysDir, sizeof szSysDir) ;
  wsprintf( szModName, "%s\\Crypt32.dll", szSysDir) ;
  hmodCrypt32 = LoadLibrary( szModName) ;
  if ( hmodCrypt32 == 0)  {
    // Crypt32.dll does not ship with the base release of OSR2.
    // If the user has it on his system, it probably comes from a
    //  version of IE.
    DWORD dwLastError = GetLastError() ;
    OSVERSIONINFO osv ;
    osv.dwOSVersionInfoSize = sizeof osv ;
    if ( ( dwLastError == ERROR_MOD_NOT_FOUND
           ||  dwLastError == ERROR_DLL_NOT_FOUND)
         &&  GetVersionEx( &osv) != FALSE
         &&  osv.dwMajorVersion == 4         // Win95, 98, ME or NT4
         &&  osv.dwMinorVersion == 0         // Win95 or NT4
         &&  osv.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS)  {// Win95
      return true ;
    }  // if OSR2 is missing Crypt32.dll
    // no valid reason for the call to fail
    ErrorLoad( szModName) ;                  // no return
  }  // if the Crypt32.dll load failed
  // successful return
  return true ;
}  // LoadCrypt32
/* function OpenCertificateStores:
   Open the system certificate stores
   We use these to validate certificates that fail validation in the
    certificate store created from the signature that contains them.
   The reason for the __try / __except block here is obscure.  Cert-
    OpenSystemStore is the first call this DLL makes to Crypt32.dll.
    Since this module may need to run in Win95 OSR2 versions lacking
    Crypt32, we cannot statically link that DLL, nor can other Zone
    modules.  Since we rarely test on old Win95 versions, sometimes
    we forget this constraint.  Depending on a load order we do not
    carefully control, we may reach this function with the statically
    loaded Crypt32 uninitialized.  This causes CertOpenSystemStore to
    fault or freeze.  We traced a fault case to a CryptGetOIDFunction-
    Address call inside CertOpenSystemSystemStore, that used a zero
    argument.  The argument came from an global variable that should
    have been initialized by a call to CryptInitOIDFunctionSet, but in
    The Twilight Zone the latter function is never called.
   We previous tried checking for a statically loaded Crypt32 before
    we dynamically load it, but that turned out to be a bad idea (bug
    11008, perhaps also 11012), since there are valid cases where
    Crypt32 is statically loaded.
*/
static bool OpenCertificateStores()
{
  // bail out if Crypt32.dll not loaded
  if ( hmodCrypt32 == 0)
    return true ;
  // loop through the canned names
```

```
  // Perhaps it would be better to enumerate the keys in
  //  HKEY_CURRENT_USER\Software\Microsoft\SystemCertificates
  // and
  //  HKEY_LOCAL_SYSTEM\Software\Microsoft\SystemCertificates
  // the latter in case we run as a service in NT4.
  int ndxSysStore
    , ndxHandle = 0
    ;
  __try  {                                    // __try / __except
    for ( ndxSysStore = 0 ; ndxSysStore < nbrSysStores ; ndxSysStore++)
 {
      HCERTSTORE hStore = CertOpenSystemStore( 0, pszSysStore[ ndxSysSto
re]) ;
      if ( hStore != 0)
        hSysStore[ ndxHandle++] = hStore ;
    }  // loop once for each system store
  }  // __try
  __except( EXCEPTION_EXECUTE_HANDLER)  {
    LogSecError( "%u %s"
               , ERROR_SECURE_POSS_CRYPT32_STAT_LINK
               , pszSysStore[ ndxSysStore]
               ) ;
    return false ;
  }  // __except
  // successful return
  return true ;
}  // OpenCertificateStores
/* function TerminateCertificates:
   Static object destruction
   Avoid memory leaks.
*/
static void __cdecl TerminateCertificates()
{
  // nothing to do if Crypt32.dll was not loaded
  if ( hmodCrypt32 == 0)
    return ;
  // close the well-known system certificate stores
  CloseCertificateStores() ;
  // unload the DLL
  FreeLibrary( hmodCrypt32) ;
}  // TerminateCertificates
// GetProcAddress.cpp
// Copyright (c) 2004. Zone Labs, LLC All Rights Reserved.
/* File GetProcAddress.cpp:
   Implement our own version of GetProcAddress
   All modules with secure static imports use this function instead
    of KERNEL32!GetProcAddress.  They reach this function via exported
    function SecureGetProcAddress in SecureAPI.cpp, imported by other
    modules as GetProcAddress (thanks to SecurePE.exe).
   Until we decide whether we can risk breaking Nortel's exthook.dll,
    we route some calls through KERNEL32!GetProcAddress.  The calls
    exthook.dll cares about are
    Name                Ordinal
    connect                   4
    gethostbyname            52
    WSAAsyncGetHostByName   103
*/
// Pre-compiled header files, must come first
#include "VSInit_pch.h"
#pragma hdrstop
// Application header files
#include <vsinit.h>
#include "vsinit_int.h"
// Temporary measure, we hope
// #define KLUGE_FOR_NORTEL_VPN 1
#ifdef KLUGE_FOR_NORTEL_VPN
#undef GetProcAddress
#endif
```

```
// Local data
// These are ASCIIZ strings XORed with x55.  This will keep an
//  attacker from easily finding the string constants we use to
//  locate the OS's GetProcAddress.
const BYTE bKERNEL32[]
  = { 0x1E, 0x10, 0x07, 0x1B, 0x10, 0x19, 0x66, 0x67, 0x55} ;
const BYTE bGetProcAddress[]
  = { 0x12, 0x30, 0x21, 0x05, 0x27, 0x3A, 0x36, 0x14
    , 0x31, 0x31, 0x27, 0x30, 0x26, 0x26, 0x55
    } ;
typedef FARPROC ( WINAPI * GETPROCADDRESS)( HMODULE, LPCSTR) ;
static GETPROCADDRESS pGetProcAddress ;
// Local functions
static FARPROC GetProcNotFound() ;
/* function InitializeGetProcAddress:
   Initialize static objects
   This function is called from the carefully sequenced static object
    initialization in VSInit.cpp.
   We retrieve the address of KERNEL32!GetProcAddress, for later use in
    processing GetProcAddress requests for forwarded functions.  By
    getting this address directly from KERNEL32, and by doing it only
    once, we resist most hooking attempts.  The determined attacker, who
    doesn't want to analyze our code, will need to patch the
    GetProcAddress entry point in KERNEL32 in order to hook our calls.
    And when he does, he'll find we call only for a forwarded export.
   Our technique assumes KERNEL32!GetProcAddress is not forwarded.  If
    it is forwarded in a later version of the OS, the call through the
    uninitialized pGetProcAddress will fault.
   To frustrate attackers further, we build the target module and
    function names from non-ASCII data.
*/
bool InitializeGetProcAddress()
{
  // get the KERNEL32 module handle
  const int lenModName = sizeof bKERNEL32 ;
  const BYTE bKeyModName = bKERNEL32[ lenModName - 1] ;
  char szModName[ lenModName] ;
  int ndxChar ;
  for ( ndxChar = 0 ; ndxChar < lenModName ; ndxChar++)
    szModName[ ndxChar] = bKERNEL32[ ndxChar] ^ bKeyModName ;
  HMODULE hmodKernel32 = GetModuleHandle( szModName) ;
  if ( hmodKernel32 == 0)                    // should never happen
    return false ;                           // error return
  // now find the function pointer
  const int lenFuncName = sizeof bGetProcAddress ;
  const BYTE bKeyFuncName = bGetProcAddress[ lenFuncName - 1] ;
  char szFuncName[ lenFuncName] ;
  for ( ndxChar = 0 ; ndxChar < lenFuncName ; ndxChar++)
    szFuncName[ ndxChar] = bGetProcAddress[ ndxChar] ^ bKeyFuncName ;
  pGetProcAddress
    = GETPROCADDRESS( MyGetProcAddress( hmodKernel32, szFuncName)) ;
  // successful return
  return true ;
}  // InitializeGetProcAddress
/* function MyGetProcAddress:
   DIY version of GetProcAddress, resists import table hooking by an
    attacker
   We punt to the real GetProcAddress if we find a forwarder RVA,
    but our technique for calling the real GetProcAddress is also
    somewhat resistant to interception.
   A determined application can still find the real GetProcAddress
    by calling GetProcAddress for "GetProcAddress".  We don't have
    any reason to block this yet, but if we did we could compare
    the function pointer we are about to return against the
    pGetProcAddress pointer we retrieved during initialization,
    substituting our function if they match.  A good reason not to
    substitute our function is that our DLL may later be unloaded,
    and a call to our function will then fault.
```

```
   Our choice of name lets our secure import stub satisfy linker
    references in the importer for the real GetProcAddress.  The effect
    is to divert all of the importer's calls to GetProcAddress to us,
    without any changes to the importer's source.
*/
FARPROC MyGetProcAddress( HMODULE hmod, LPCSTR lpProcName)
{
#ifdef KLUGE_FOR_NORTEL_VPN
  return GetProcAddress( hmod, lpProcName) ;
#else
  // the caller has already validated lpProcName
  // now do the work
  __try  {                                    // __try / __except
    // find the export directory
    // Since the module has already been successfully loaded by the OS,
    //  we assume no further validation is necessary.
    PBYTE pbMod = PBYTE( hmod) ;
    PIMAGE_DOS_HEADER pHdrDOS = PIMAGE_DOS_HEADER( hmod) ;
    PIMAGE_NT_HEADERS32 pHdrNT
      = PIMAGE_NT_HEADERS32( pbMod + pHdrDOS->e_lfanew) ;
    PIMAGE_OPTIONAL_HEADER32 pHdrOpt = &pHdrNT->OptionalHeader ;
    PIMAGE_DATA_DIRECTORY pDirExport
      = pHdrOpt->DataDirectory + IMAGE_DIRECTORY_ENTRY_EXPORT ;
    DWORD dwExpDirRVA = pDirExport->VirtualAddress
        , dwExpDirSize = pDirExport->Size
        ;
    PIMAGE_EXPORT_DIRECTORY pExpDir
      = PIMAGE_EXPORT_DIRECTORY( pbMod + dwExpDirRVA) ;
    // macros to simplify conversions
    // The RVAToPtr result must be cast to the appropriate data type.
    #define RVAToPtr(a) PVOID( PBYTE( pExpDir) + (a) - dwExpDirRVA)
    #define RVAToString(a) PSTR( RVAToPtr(a))
    #define RVAToPFn(a) FARPROC( RVAToPtr(a))
    #define RVAToPDWord(a) PDWORD( RVAToPtr(a))
    #define RVAToPWord(a) PWORD( RVAToPtr(a))
    #define PtrToRVA(a) DWORD( ( PBYTE(a) - PBYTE( pExpDir)) + dwExpDirR
VA)
    // macro to decide if an export is forwarded
    #define IsThisExportForwarded(a) DWORD(a) - dwExpDirRVA < dwExpDirSi
ze
    // extract some information from the export directory
    PDWORD pdwAddrOfFuncs = RVAToPDWord( pExpDir->AddressOfFunctions) ;
    const int nbrFuncs = pExpDir->NumberOfFunctions ;
    DWORD dwBaseOrd = pExpDir->Base ;        // base ordinal
    PDWORD pdwAddressOfNames = RVAToPDWord( pExpDir->AddressOfNames) ;
    PWORD pwAddrOfNameOrds = RVAToPWord( pExpDir->AddressOfNameOrdinals)
 ;
    // handle ordinals first, since this is simpler
    DWORD dwRVAFunc ;
    if ( DWORD( lpProcName) < 65536)  {
      DWORD dwOrdinal = DWORD( lpProcName) ;
      int ndxFunc = dwOrdinal - dwBaseOrd ;
      if ( ndxFunc >= nbrFuncs)  {
        SetLastError( ERROR_INVALID_ORDINAL) ;
        return 0 ;
      }  // if an invalid ordinal
      dwRVAFunc = pdwAddrOfFuncs[ ndxFunc] ;
      if ( IsThisExportForwarded( dwRVAFunc))
        return pGetProcAddress( hmod, lpProcName) ;
      else
        return RVAToPFn( dwRVAFunc) ;
    }  // if an ordinal request
    // import by name, resolved by a binary search of the names array
    // The _ is treated as any other character, suggesting Win32 uses
    //  a string sort, not a word sort.  So we must use strcmp, not
    //  lstrcmp.
    int ndxLo = 0
      , ndxHi = pExpDir->NumberOfNames - 1
```

```
      , ndxTry
      , nResult
      ;
    // bail out if there are no names
    if ( ndxHi < 0)
      return GetProcNotFound() ;
    // compare with the low end
    ndxTry = ndxLo ;
    nResult = strcmp( lpProcName, RVAToString( pdwAddressOfNames[ ndxTry
])) ;
    if ( nResult == 0)  {                    // if the first string
      dwRVAFunc = pdwAddrOfFuncs[ pwAddrOfNameOrds[ ndxTry]] ;
      if ( IsThisExportForwarded( dwRVAFunc))
        return pGetProcAddress( hmod, lpProcName) ;
      else
        return RVAToPFn( dwRVAFunc) ;
    }  // if the first string
    if ( nResult < 0)                        // if below all the strings
      return GetProcNotFound() ;
    // compare with the high end
    ndxTry = ndxHi ;
    nResult = strcmp( lpProcName, RVAToString( pdwAddressOfNames[ ndxTry
])) ;
    if ( nResult == 0)  {                    // if the last string
      dwRVAFunc = pdwAddrOfFuncs[ pwAddrOfNameOrds[ ndxTry]] ;
      if ( IsThisExportForwarded( dwRVAFunc))
        return pGetProcAddress( hmod, lpProcName) ;
      else
        return RVAToPFn( dwRVAFunc) ;
    }  // if the last string
    if ( nResult > 0)                        // if above all the strings
      return GetProcNotFound() ;
    // now do the binary search
    // At each iteration the high and low values have been tested, so
    //  the search ends in failure when there are no values between
    //  them to test.
    int nLimit = 0 ;
    while ( ndxHi - ndxLo > 1)  {            // while range has an untrie
d value
      if ( nLimit++ > 20) break ;
      ndxTry = ( ndxLo + ndxHi) / 2 ;        // rounds down
      nResult = strcmp( lpProcName, RVAToString( pdwAddressOfNames[ ndxT
ry])) ;
      if ( nResult == 0)  {                  // if a match
        dwRVAFunc = pdwAddrOfFuncs[ pwAddrOfNameOrds[ ndxTry]] ;
        if ( IsThisExportForwarded( dwRVAFunc))
          return pGetProcAddress( hmod, lpProcName) ;
        else
          return RVAToPFn( dwRVAFunc) ;
      }  // if a match
      if ( nResult > 0)                      // if target is above the gu
ess
        ndxLo = ndxTry ;                     // move the bottom up
      else                                   // if target is below the gu
ess
        ndxHi = ndxTry ;                     // move the top down
    }  // loop until the range has one index
    return GetProcNotFound() ;
  }  // __try
  __except( EXCEPTION_EXECUTE_HANDLER)  {
    return GetProcNotFound() ;
  }  // __except
#endif
}  // MyGetProcAddress
/* function GetProcNotFound:
   Coding aid to reduce clutter in VSGetProcAddress
*/
static FARPROC GetProcNotFound()
```

```
{
  SetLastError( ERROR_PROC_NOT_FOUND) ;
  return 0 ;
}  // GetProcNotFound
// GetProcImport.asm
// Copyright (c) 2004. Zone Labs, LLC All Rights Reserved.
COMMENT *
File GetProcImport.asm:
Spoof an import reference to avoid a static link to GetProcAddress
This is part of our program to eliminate static links to GetProc-
Address, a deterrent to some attacks.
In VSInit.dll, we replace GetProcAddress calls in our source by calls to
our custom SecureGetProcAddress, and we enforce this with a macro that
busts usage of GetProcAddress.  But a developer can inadvertently build
without this macro, or can link a pre-built library containing a call.
In particular, we use delayed loading of Crypt32.dll since the DLL is
not present on Win95 OSR2 without IE4.  The delay loader helper calls
GetProcAddress.  We could rebuild the helper (DelayHlp.cpp in the Visual
Studio VC98\Include directory), but we prefer not to.  So we must
satisfy a call through data variable
  __imp__GetProcAddress@8
normally done via an import definition in Kernel32.lib.
In this file we spoof the definition to avoid the import.  We can create
the desired symbol in C, but only as a function name, and this evidently
does not keep the linker from using the import.
*
        .586
        .MODEL   FLAT, C
        .CODE
         EXTERN  SecureGetProcAddress@8:NEAR
         PUBLIC  _imp__GetProcAddress@8
        .DATA
_imp__GetProcAddress@8 DWORD SecureGetProcAddress@8
         END
// SecureAPI.cpp
// Copyright (c) 2004. Zone Labs, LLC All Rights Reserved.
/* File SecureAPI.cpp:
   Functions used to implement secure static links and dynamic links
   Deadlock warning:  GetModuleFileName uses the OS's per-process
    critical section.  This is the same critical section the OS holds
    for the duration of a DLL's initialization and termination.  We also
    use various private critical sections.  Many of our functions are
    called during DLL initialization.  To avoid deadlock in functions
    called on different threads, we must always request critical
    sections in the same order.  So any function that can be called on a
    thread not holding the per-process critical section must ensure it
    does not hold any private critical section when calling GetModule-
    FileName and GetModuleHandle, among other functions.
*/
// Pre-compiled header files, must come first
#include "VSInit_pch.h"
#pragma hdrstop
// Compiler header files
#include <stdlib.h>
#include <stdio.h>
#include <map>
#include <set>
// Application header files
#include <vsinit.h>
#include "vsinit_int.h"
// Data structures
typedef FARPROC ( WINAPI * PFNGETPROCADDRESS)( IN HMODULE hModule
                                              , IN LPCSTR lpProcName
                                              ) ;
class DYNLINK {
public:
  HMODULE hmod ;                              // module handle
  PFNGETPROCADDRESS pGetProcAddress ;         // hooked function
```

```
  char * pszName ;                              // full module file path nam
e
  PDWORD pdwPatchTarget ;                       // patch target in IAT
  BYTE bHookGetProcAddress[ 12] ;               // multiple of 4 for alignme
nt
  DYNLINK() : pszName( 0)
            , hmod( 0)
            , pdwPatchTarget( 0)
            { bHookGetProcAddress[ 0] = 0x58 ;// POP EAX
              bHookGetProcAddress[ 1] = 0x68 ;// PUSH imm-32
              bHookGetProcAddress[ 6] = 0x50 ;// PUSH EAX
              bHookGetProcAddress[ 7] = 0xe9 ;// JMP NEAR32
            }
  ~DYNLINK() { if ( pszName != 0)
                 delete [] pszName ;
             }
} ;
typedef DYNLINK * PDYNLINK ;
static std::set<PDYNLINK> * psetDynLink = 0 ;// track PDYNLINK pointers
class VALIDATEDMODULE {
public:
  char * pszName ;                              // fully qualified path name
  DWORD dwHash ;                                // of pszName
  DWORD nbrExports ;                            // number of secure exports
  DWORD dwModBaseAddr ;                         // this module's base addres
s
  PDWORD pdwRVAExports ;                        // RVAs for secure exports
  PDWORD pdwHashExports ;                       // hash values for exported
names
  DWORD dwAddrSecureStaticLink ;                // module's static link funn
el
  PDYNLINK pDynLink ;                           // dynamic link hook
  CRITICAL_SECTION cs ;                         // serialize access
  std::set<DWORD> noPatch ;                     // addresses where patches f
ailed
  VALIDATEDMODULE() : dwModBaseAddr( 0)
                    , pszName( 0)
                    , dwAddrSecureStaticLink( 0)
                    , pDynLink( 0)
                    { InitializeCriticalSection( &cs) ; }
  ~VALIDATEDMODULE() { DeleteCriticalSection( &cs) ;
                       if ( pszName != 0)
                         delete [] pszName ;
                       if ( pDynLink != 0)  {
                         psetDynLink->erase( pDynLink) ;// stop tracking
 it
                         delete pDynLink ;
                         pDynLink = 0 ;
                       }                        // if there is a dynamic lin
k hook
                     }
} ;
typedef VALIDATEDMODULE * PVALIDATEDMODULE ;
static std::multimap<DWORD,PVALIDATEDMODULE> * pmapValMod = 0 ;
typedef struct _IMPORTTBL {                     // matches structure in Secu
rePE.cpp
  PVALIDATEDMODULE pValMod ;                    // for this imported module
                                                // the next field must be la
st
  char     szModName[ 1] ;                      // variable length
} IMPORTTBL, * PIMPORTTBL ;
// Constants
#define MAX_NORMAL_ORDINAL 65535                // so two high bytes are zer
o
#define MAX_SECURE_ORDINAL  2047                // must match SecurePE.cpp
const DWORD dwLargePrime = ( 1u << 31) - 1 ;// 8th Mersenne prime
// The compiler emits bloated code for aggregate constants (e.g., arrays
//  or structures) in function scope.
```

```
const char szKernel32[] = "KERNEL32.dll"
         , szGetProcAddress[] = "GetProcAddress"
         , szDummyImport[] = "SUVW"          // SUVW() is in this file
         , szUnknown[] = "(Unknown)"
         ;
// Global data
extern HMODULE g_hModThis ;                  // this DLL's module handle
// Local data
static CRITICAL_SECTION csTable ;            // access to the STL map
static CRITICAL_SECTION csDynLinkHook ;      // set a GetProcAddress hook
static PDWORD pCRCTable ;                    // used to hash function nam
es
static PVALIDATEDMODULE pValModBase ;
static PVALIDATEDMODULE pValModThis = 0 ;    // expect same as pValModBas
e
#define MAX_MODULES 40
static int nbrModules = 0 ;
// Static functions called from other modules (via trickery)
static FARPROC WINAPI ResolveDynamicLink( PDYNLINK pDynLink
                                        , HMODULE hmodTarget
                                        , LPCSTR pProcName
                                        ) ;
static FARPROC WINAPI ResolveStaticLink( PVALIDATEDMODULE pValModCaller
                                       , PIMPORTTBL pTable
                                       ) ;
// Local functions
static int __cdecl CompareDWORD( const void *e1, const void *e2) ;
static PVALIDATEDMODULE FindTableSlot( HMODULE hmod, bool fAddIfNotFound
) ;
static FARPROC WINAPI FixupStaticLinkFunnel( PVALIDATEDMODULE pValModNot
Yet
                                           , PIMPORTTBL pTable
                                           ) ;
static const char * GetCallerFileName( DWORD dwRetAddr
                                     , char * pszModNameCaller
                                     ) ;
static FARPROC GetProcAddressExp( PVALIDATEDMODULE pValMod
                                , DWORD dwHash
                                , DWORD dwHint
                                ) ;
static int GetSizeOfPushInstruction( DWORD dwPushed
                                   , DWORD dwAddrAfterPush
                                   , PVALIDATEDMODULE pValMod
                                   ) ;
static DWORD HashString( LPCSTR pszString) ;
static bool IsModulePatched( PVALIDATEDMODULE pValMod) ;
static bool MyIsBadStringPointer( LPCTSTR lpsz, UINT_PTR ucchMax) ;
static DYNLINK * newDYNLINK() ;
static bool ParseSecureExports( PVALIDATEDMODULE pValMod) ;
static PVALIDATEDMODULE WINAPI SecureLoadLibrary( LPCSTR lpFileName) ;
static PDYNLINK SetTheGetProcAddressHook( HMODULE hmod) ;
static void __cdecl TerminateSecureAPI() ;
static bool UndoTheGetProcAddressHook( const DYNLINK * pDynLink) ;
static PVALIDATEDMODULE ValidatedModuleFromCodeAddress( PVOID pvAddr) ;
/* function AreSecureDynLinksAllowed:
   If they are, patch the calling module to enable them
   A module must pass our validation test, but we cast a wide net.  The
    valid module can be the caller, or the current EXE, or any of its
    parent EXEs.  This net may be too wide, since it allows access to
    all secure APIs.  We might to restrict the APIs available based on
    which module we validate, but this will add an administrative
    burden.
*/
bool WINAPI AreSecureDynLinksAllowed( HMODULE hmod)
{
  // we assume the caller has validated his caller, which may be
  //  unsigned, by other means
  // For the OEM API we may need to revisit the caller's validation.
```

```
  // Here we validate the process' EXE chain.
  if ( IsMyProcessOrAnAncestorCertified( SAPI_LOG_TVDEBUG) == false)
    return false ;
  SetTheGetProcAddressHook( hmod) ;
  return true ;
}  // AreSecureDynLinksAllowed
/* function MSIPrepareSecureApi:
   patch the calling module to enable them
*/
bool WINAPI MSIPrepareSecureApi( HMODULE hmod)
{
  SetTheGetProcAddressHook( hmod) ;
  return true ;
}  // MSIPrepareSecureApi
/* function InitializeSecureAPI:
   Initialize static objects
   Prepare data structures for module validation.
   This function is called from the carefully sequenced static object
    initialization in VSInit.cpp.
*/
bool InitializeSecureAPI()
{
  // initialize the critical sections
  // The STL map of PVALIDATEDMODULE pointers is not thread-safe.
  InitializeCriticalSection( &csTable) ;
  // Avoid a race when setting a module's GetProcAddress hook.
  InitializeCriticalSection( &csDynLinkHook) ;
  // allocate the tables that track validated modules
  pValModBase = new VALIDATEDMODULE[ MAX_MODULES] ;
  if ( pValModBase == 0)                    // if allocation failed
    LogSecFatal( "%u %u", ERROR_SECURE_ALLOCATION_1) ;
  // allocate and generate the CRC-32 table
  // We can save 1 KB, plus this table generation code, by pre-defining
  //  the CRC-32 table in a shared data segment.  The only drawback is
  //  that the table is then visible in a disassembly.
  const int TABLE_SIZE = 256 ;
  const int nbrBytes = TABLE_SIZE * sizeof( DWORD) ;
  pCRCTable = PDWORD( HeapAlloc( GetProcessHeap(), 0, nbrBytes)) ;
  if ( pCRCTable == 0)                      // if allocation failed
    LogSecFatal( "%u %u", ERROR_SECURE_NOT_ENOUGH_MEMORY_2, nbrBytes) ;
  // populate the table
  const DWORD Polynomial = 0xedb88320 ;
  for ( int ndxOuter = 0 ; ndxOuter < TABLE_SIZE ; ndxOuter++)  {
    DWORD crc = ndxOuter ;
    for ( int ndxInner = 0 ; ndxInner < 8 ; ndxInner++)  {
      if ( ( crc & 1) != 0)
        crc = (crc >> 1) ^ Polynomial ;
      else
        crc >>= 1 ;
    }
    pCRCTable[ ndxOuter] = crc ;
  }  // loop once for each table slot
  // create the multimap
  // We cannot declare the multimap as a static object, since its
  //  initializer may not have run before we first use it, causing an
  //  access exception.
  // We should check for a nonzero return, but we wouldn't know how
  //  to handle an error in an initializer.  Let's hope this call
  //  always succeeds.
  pmapValMod = new std::multimap<DWORD,PVALIDATEDMODULE> ;
  // create the dynamic link tracking set
  // We cannot declare the set as a static object, since its destructor
  //  may run before TerminateSecureAPI, causing references to it in
  //  that function to fault.
  psetDynLink = new std::set<PDYNLINK> ;
  // set the termination function
  atexit( TerminateSecureAPI) ;
  // successful return
```

```
  return true ;
}  // InitializeSecureAPI
/* function IsModuleValid:
   As it says
   This validates another module and prepares that module to call
    secure static and dynamic links.  The call to this function origi-
    nates in an exception deliberately raised during the subject
    module's initialization.
   The validation itself is done in IsPEFileValid.
   Open:  Validate the patch target before patching.  Ensure it is
           in the module, and that it contains a virgin pattern.
*/
bool IsModuleValid( HMODULE hmod, PVALIDATESELF pvalSelf)
{
  // get the EXE's file name
  // We use this only in fatal error messages, but we retrieve it now
  //  to avoid the risk of deadlock.  See the file prolog for more
  //  information.
  char szMyEXE[ MAX_PATH] ;
  GetModuleFileName( 0, szMyEXE, MAX_PATH) ;
  // find a table slot
  PVALIDATEDMODULE pValMod = FindTableSlot( hmod, true) ;
  CCritSecAutoRelease csAuto( &pValMod->cs) ;// serialize access
  // have we previously validated and adjusted this module?
  if ( IsModulePatched( pValMod))           // if yes
    return true ;                           // no need to do it again
  // validate the file
  if ( IsPEFileValid( pValMod->pszName, SAPI_LOG_TVDEBUG) )  {
    // set the module handle
    pValMod->dwModBaseAddr = DWORD( hmod) ;
    ParseSecureExports( pValMod) ;          // parse the secure export t
able
    // hook the module's GetProcAddress call if this is not VSInit
    // We track the dynamic link hook here only to reduce memory
    //  leakage.  If we develop a general anti-leak technique for
    //  the DYNLINK blocks, e.g., by hooking each DLL's entry RVA,
    //  we should remove this pointer or else we may find ourselves
    //  deleting a stale pointer.
    if ( hmod != g_hModThis)  {
      PDYNLINK pDynLink = SetTheGetProcAddressHook( hmod) ;
      if ( pValMod->pDynLink != 0)  {       // if a prior DynLink hook
        psetDynLink->erase( pValMod->pDynLink) ;// stop tracking it
        delete pValMod->pDynLink ;          // delete it
      }
      pValMod->pDynLink = pDynLink ;        // set the new DynLink hook
    }  // if not VSInit
    // patch the static link resolver in the calling module so that
    //  it jumps to VSInit to patch each static link funnel on the first
    //  call to that funnel
    // The dynamic link resolver is initialized via a secure static
    //  link in the caller.  For OEM callers forbidden to use static
    //  links to our modules, we will need addtional processing,
    //  probably called from tvInitializeEx.
    // We optimistically neglect to validate the patch address.  If it
    //  is wrong, bad things will happen soon.
    PatchDWord( pvalSelf->dwAddrPatchResolveStatic + 1
              , DWORD( &FixupStaticLinkFunnel)
                - ( pvalSelf->dwAddrPatchResolveStatic + 5)
              ) ;
    // save the starting address of the module's SecureStaticLink
    // Do this last, so that we can use it later to test whether self-
    //  validation and patching are already complete.
    pValMod->dwAddrSecureStaticLink = pvalSelf->dwAddrPatchResolveStatic
 ;
    // successful return
    return true ;
  }  // if validation succeeded
  // validation failed
```

```
  // We still hold the csTable critical section, which should avoid
  //  multiple concurrent message boxes for this process.
  // display an error message
  // To avoid overloading the user with bad news, we report only one
  //  validation error per process, then force a process exit.
  // If preparing a fancy message causes a stack overflow, we'll feel
  //  very silly.
  char szMsg[ MAX_PATH + 100] ;
  sprintf( szMsg, "Validation failed for %s.", pValMod->pszName) ;
  MessageBox( 0
            , szMsg
            , szMyEXE
            , MB_OK | MB_ICONSTOP | MB_TASKMODAL | MB_TOPMOST
            ) ;
  // write to the debug log and die
  LogSecFatal( "%u src=%s trg=%s"
             , ERROR_SECURE_SELF_VALIDATION
             , szMyEXE
             , pValMod->pszName
             ) ;
  // avoid a compiler warning
  // The LogSecFatal call keeps us from reaching here.
  return false ;
}  // IsModuleValid
/* function ResolveDynamicLink:
   Call GetProcAddress on behalf of a self-validated module that has
    one or more secure imports
   This function is called from other modules, but its address is not
    exposed to the linker.
*/
static FARPROC WINAPI ResolveDynamicLink( PDYNLINK pDynLink
                                        , HMODULE hmodTarget
                                        , LPCSTR pProcName
                                        )
{
  // consider validating the DYNLINK block to reduce the chance of
  //  our being spoofed by an attacker
  // guard against a bogus pointer
  DWORD dwOrdinal = DWORD( pProcName) ;     // in case of an ordinal req
uest
  if ( dwOrdinal > MAX_NORMAL_ORDINAL)  {   // if a string pointer
    const UINT_PTR BIGGEST_PROC_NAME = 64 ; // big enough?
    if ( MyIsBadStringPointer( pProcName, BIGGEST_PROC_NAME))  {
      SetLastError( ERROR_INVALID_PARAMETER) ;
      return 0 ;                            // error return
    }
  }  // if pProcName is a string pointer
#if 0                                       // testing only
  // trace all dynamic link calls
  if ( dwOrdinal <= MAX_NORMAL_ORDINAL)  {
    char szModFileName[ MAX_PATH] ;
    GetModuleFileName( hmodTarget, szModFileName, MAX_PATH) ;
    LogSecError( "%u %s to %s %u"
               , INFO_SECURE_DYNAMIC_LINK
               , pDynLink->pszName
               , szModFileName
               , dwOrdinal
               ) ;
  }
  else  {
    char szModFileName[ MAX_PATH] ;
    GetModuleFileName( hmodTarget, szModFileName, MAX_PATH) ;
    LogSecError( "%u %s to %s %s"
               , INFO_SECURE_DYNAMIC_LINK
               , pDynLink->pszName
               , szModFileName
               , pProcName
               ) ;
```

```
  }
#endif
  // call the hooked function in the caller's module
  FARPROC pfnResult = pDynLink->pGetProcAddress( hmodTarget, pProcName)
;
  // continue only if the function failed with a specific error code
  if ( pfnResult != 0)
    return pfnResult ;
  switch ( GetLastError())  {
    case ERROR_PROC_NOT_FOUND :
    case ERROR_INVALID_ORDINAL :
      break ;                                 // it may be a secure API
    default :
      return 0 ;                              // error return
  }  // switch on the error code
  // this may be a secure export
  // find the target module's block
  // GetProcAddressExp ensures the block is not stale.
  PVALIDATEDMODULE pValModCallee ;
  if ( hmodTarget == g_hModThis)
    pValModCallee = pValModThis ;
  else
    pValModCallee = FindTableSlot( hmodTarget, false) ;
  if ( pValModCallee == 0)  {
    SetLastError( ERROR_MOD_NOT_FOUND) ;
    return 0 ;                                // error return
  }
  CCritSecAutoRelease csAuto( &pValModCallee->cs) ;// serialize access
  // process a by-ordinal request
  if ( dwOrdinal <= MAX_SECURE_ORDINAL)  {  // if an ordinal
    pfnResult = GetProcAddressExp( pValModCallee, dwOrdinal, DWORD( -1))
 ;
    if ( pfnResult != 0)                      // if successful
      return pfnResult ;                      // return the function point
er
    LogSecError( "%u %s %s %u"
               , ERROR_SECURE_DYN_LINK_ORD
               , pDynLink->pszName
               , pValModCallee->pszName
               , dwOrdinal
               ) ;
    SetLastError( ERROR_PROC_NOT_FOUND) ;
    return 0 ;                                // error return
  }  // if pProcName is not a string pointer
  // import by name
  // search the target's secure export tables
  DWORD dwHashVal = HashString( pProcName) ;
  if ( dwHashVal <= MAX_SECURE_ORDINAL)  {
    LogSecError( "%u %s %s %s %u"
               , ERROR_SECURE_DYN_LINK_HASH
               , pDynLink->pszName
               , pValModCallee->pszName
               , pProcName
               , dwHashVal
               ) ;
    SetLastError( ERROR_PROC_NOT_FOUND) ;
    return 0 ;                                // error return
  }  // if the name hashed to an embargoed ordinal
  pfnResult = GetProcAddressExp( pValModCallee, dwHashVal, DWORD( -1)) ;
  if ( pfnResult != 0)                        // if successful
    return pfnResult ;                        // return the function point
er
  LogSecError( "%u %s %s %s"
             , ERROR_SECURE_DYN_LINK_NAME
             , pDynLink->pszName
             , pValModCallee->pszName
             , pProcName
             ) ;
```

```
  SetLastError( ERROR_PROC_NOT_FOUND) ;
  return 0 ;                                    // error return
}  // ResolveDynamicLink
/* function ResolveStaticLink:
   Resolve a static link, back patching the caller if possible
   This function is called from other modules, but its address is not
    exposed to the linker.
*/
static FARPROC WINAPI ResolveStaticLink( PVALIDATEDMODULE pValModCaller
                                       , PIMPORTTBL pTable
                                       )
{
  // retrieve forbidden fruit from the stack
  DWORD dwAddrFunnelRet = * ( PDWORD( &pValModCaller) - 1)// return in t
he funnel
      , dwAddrStubRet   = * ( PDWORD( &pTable) + 9)
      , dwHint          = * ( PDWORD( &pTable) + 10)
      , dwHash          = * ( PDWORD( &pTable) + 11)
      , dwAddrCallerRet = * ( PDWORD( &pTable) + 12)
      ;
  PCPUREGS pRegs = PCPUREGS( PDWORD( &pTable) + 1) ;
  // Open: Validate the caller against pValModCaller
  // Is this really necessary?  We know the caller must still exist,
  //  since he just called us.  The only other reason to validate is to
  //  guard against an attack, but we are already partly protected by
  //  the obscure way the ResolveStaticLink call is patched into the
  //  caller's code during self-validation.
  // validate the funnel code
  // We know almost exactly the machine instructions in the funnel,
  //  and we validate them here to screen out careless attackers.  Of
  //  course, most attackers will try dynamic links before static
  //  links.
//IsNearDirectCall( dwAddrFunnelRet, DWORD( ResolveStaticLink)) ;
  DWORD dwAddrFunnel = dwAddrFunnelRet - 16 ;
  const DWORD sizeFunnel = 24 ;
  PBYTE pbInst = PBYTE( dwAddrFunnel) ;
  if ( IsBadReadPtr( pbInst, sizeFunnel))  {
    LogSecFatal( "%u %s %X + %X"
               , ERROR_SECURE_BAD_FUNNEL
               , pValModCaller->pszName
               , pValModCaller->dwModBaseAddr
               , dwAddrFunnel - pValModCaller->dwModBaseAddr
               ) ;
  }  // if bad funnel code
  if ( pbInst[ 0] != 0x60                       // PUSHAD
       ||  pbInst[ 1] != 0x68                   // PUSH imm 4-byte value
       ||  pbInst[ 6] != 0x68                   // PUSH imm 4-byte value
    /* ||  IsAddressInThisModule( * ( PDWORD)( pbInst + 2)) == false */
)  {
    LogSecFatal( "%u %s %X + %X"
                 " %2.2X %2.2X %2.2X %2.2X %2.2X %2.2X %2.2X %2.2X"
               , ERROR_SECURE_BAD_FUNNEL
               , pValModCaller->pszName
               , pValModCaller->dwModBaseAddr
               , dwAddrFunnel - pValModCaller->dwModBaseAddr
               , pbInst[ 0], pbInst[ 1], pbInst[ 2], pbInst[ 3]
               , pbInst[ 4], pbInst[ 5], pbInst[ 6], pbInst[ 7]
               ) ;
  }  // if bad funnel code
  // validate the stub code
  // The stub has two immediate PUSHes, one for the hash value, one for
  //  the hint.  The hash is almost always a 4-byte PUSH, and the hint
  //  is usually a 1-byte PUSH.
//IsNearDirectCall( dwAddrStubRet, dwAddrFunnel) ;
  DWORD dwAddrCallFunnel = dwAddrStubRet - 5// near direct call
      , dwAddrPushHint
          = dwAddrCallFunnel - GetSizeOfPushInstruction( dwHint
                                                       , dwAddrCallFunne
```

```
l
                                                                , pValModCaller
                                                                )
      , dwAddrPushHash
          = dwAddrPushHint - GetSizeOfPushInstruction( dwHash
                                                     , dwAddrPushHint
                                                     , pValModCaller
                                                     )
      ;
  DWORD dwAddrStub = dwAddrPushHash ;
  // load the target module if we have not already done so
  // We allow a LoadLibrary race here, then back out one of the two
  //  loads if we find a race occurred.  We used to protect this code
  //  with a process-wide critical section, but this occasionally
  //  caused a deadlock (bug 11458), since this function can be called
  //  while the OS holds the _LoaderLock critical section.
  if ( pTable->pValMod == 0)  {
    PVALIDATEDMODULE pValModTarget = SecureLoadLibrary( pTable->szModNam
e) ;
    if ( pValModTarget == 0)  {
      LogSecFatal( "%u %d %s %X + %X %s"
                 , ERROR_SECURE_LOAD_LIBRARY
                 , GetLastError()
                 , pValModCaller->pszName
                 , pValModCaller->dwModBaseAddr
                 , dwAddrCallerRet - pValModCaller->dwModBaseAddr
                 , pTable->szModName
                 ) ;
    }  // if the module load failed
    // serialize access so we can detect a LoadLibrary race
    CCritSecAutoRelease csAutoCallee( &pValModTarget->cs) ;// serialize
access
    if ( pTable->pValMod != 0)              // if we lost a race
      FreeLibrary( HMODULE( pTable->pValMod->dwModBaseAddr)) ;
    else                                    // if no race, or if we won
      pTable->pValMod = pValModTarget ;     // set the table pointer
  }  // if the module was not already loaded
  // get the target VALIDATEDMODULE control block
  // GetProcAddressExp ensures the block is not stale.
  PVALIDATEDMODULE pValModCallee = pTable->pValMod ;
  CCritSecAutoRelease csAutoCallee( &pValModCallee->cs) ;// serialize ac
cess
  // find the desired function
  FARPROC pfnTarget = GetProcAddressExp( pValModCallee, dwHash, dwHint)
;
  if ( pfnTarget == 0)  {                   // if the function was not f
ound
    LogSecFatal( "%u %s %X %d %s %X + %X"
               , ERROR_SECURE_ORDINAL_UNKNOWN
               , pValModCallee->pszName
               , dwHash
               , dwHint
               , pValModCaller->pszName
               , pValModCaller->dwModBaseAddr
               , dwAddrCallerRet - pValModCaller->dwModBaseAddr
               ) ;
  }  // if the ordinal was not found
#if 0                                       // testing only
  LogSecError( "%u %s %X + %X %X %d -> %s %X + %X"
             , INFO_SECURE_STATIC_LINK
             , pValModCaller->pszName
             , pValModCaller->dwModBaseAddr
             , dwAddrCallerRet - pValModCaller->dwModBaseAddr
             , dwHash
             , dwHint
             , pValModCallee->pszName
             , pValModCallee->dwModBaseAddr
             , DWORD( pfnTarget) - pValModCallee->dwModBaseAddr
```

```
            ) ;
#endif
  // don't try back patching the caller if we have already tried this
  //  address and failed
  CCritSecAutoRelease csAutoCaller( &pValModCaller->cs) ;// serialize ac
cess
  if ( pValModCaller->noPatch.find( dwAddrCallerRet)
         != pValModCaller->noPatch.end())  {
    return pfnTarget ;                      // return target address to
caller
  }
  // try to back patch the caller
  if ( BackPatch( dwAddrCallerRet
                , DWORD( pfnTarget)
                , dwAddrStub
                , pRegs
                ) == false)  {
    pValModCaller->noPatch.insert( dwAddrCallerRet) ;
  }  // if the back patch failed
  // return the target address to the caller
  return pfnTarget ;
}  // ResolveStaticLink
/* function SUVW:
   Null function we use to test for a dynamic link hook
   If a module's GetProcAddress function can find this function, it
    is already hooked.
   We created a separate function, rather than use an existing one,
    since we don't want to inadverently expose the name of a secure
    function that currently is only statically linked.
   This name is intentionally obscure.  The four byte sequence occurs
    often in code.
*/
void WINAPI SUVW()
{
}  // SUVW
/* function SecureGetProcAddress:
   DIY version of GetProcAddress, resists import table hooking by an
    attacker, and also resolves secure dynamic links
   With a little magic, calls to GetProcAddress in both VSInit and
    in other self-validating modules go here instead of to KERNEL32!
    GetProcAddress.
*/
FARPROC WINAPI SecureGetProcAddress( HMODULE hmodTarget, LPCSTR pProcNam
e)
{
  // guard against a bogus pointer
  DWORD dwOrdinal = DWORD( pProcName) ;    // in case of an ordinal requ
est
  if ( dwOrdinal > MAX_NORMAL_ORDINAL)  {   // if a string pointer
    const UINT_PTR BIGGEST_PROC_NAME = 256 ;// big enough?
    if ( MyIsBadStringPointer( pProcName, BIGGEST_PROC_NAME))  {
      SetLastError( ERROR_INVALID_PARAMETER) ;
      return 0 ;                            // error return
    }
  }  // if pProcName is a string pointer
#if 0                                       // testing only
  // trace all dynamic link calls
  char szSrcFileName[ MAX_PATH]
     , szTrgFileName[ MAX_PATH]
     ;
  GetCallerFileName( * ( PDWORD( &hmodTarget) -1), szSrcFileName) ;
  GetModuleFileName( hmodTarget, szTrgFileName, MAX_PATH) ;
  if ( dwOrdinal <= MAX_NORMAL_ORDINAL)  {
    LogSecError( "%u %s to %s %u"
               , INFO_SECURE_DYNAMIC_LINK
               , szSrcFileName
               , szTrgFileName
               , dwOrdinal
```

```
                ) ;
    }
    else  {
      LogSecError( "%u %s to %s %s"
                , INFO_SECURE_DYNAMIC_LINK
                , szSrcFileName
                , szTrgFileName
                , pProcName
                ) ;
    }
#endif
    // look for a conventional import first
    FARPROC pfnResult = MyGetProcAddress( hmodTarget, pProcName) ;
    // continue only if the lookup failed with a specific error code
    if ( pfnResult != 0)
      return pfnResult ;
    switch ( GetLastError())  {
      case ERROR_PROC_NOT_FOUND :
      case ERROR_INVALID_ORDINAL :
        break ;                                 // it may be a secure API
      default :
        return 0 ;                              // error return
    }  // switch on the error code
    // this may be a secure export
    // find the target module's block
    // GetProcAddressExp ensures the block is not stale.
    PVALIDATEDMODULE pValModCallee = FindTableSlot( hmodTarget, false) ;
    if ( pValModCallee == 0)  {
      SetLastError( ERROR_MOD_NOT_FOUND) ;
      return 0 ;                                // error return
    }
    // serialize access to the target module
    // We save the callee's module name first, so that we can use it in
    //  error messages after releasing the module's block.  We release
    //  the module's block first, so that we can call GetModuleFileName
    //  in those paths without risk of critical section deadlock.  Get-
    //  ModuleFileName requires the OS's LoaderLock critical section.
    EnterCriticalSection( &pValModCallee->cs) ;// serialize access
    char szModNameCallee[ MAX_PATH] ;
    lstrcpy( szModNameCallee, pValModCallee->pszName) ;
    // process a by-ordinal request
    if ( dwOrdinal <= MAX_SECURE_ORDINAL)  {  // if an ordinal
      pfnResult = GetProcAddressExp( pValModCallee, dwOrdinal, DWORD( -1))
 ;
      LeaveCriticalSection( &pValModCallee->cs) ;
      if ( pfnResult != 0)                      // if successful
        return pfnResult ;                      // return the function point
er
      char szModNameCaller[ MAX_PATH] ;
      LogSecError( "%u %s %s %u"
                , ERROR_SECURE_DYN_LINK_ORD
                , GetCallerFileName( * ( PDWORD( &hmodTarget) -1)
                                   , szModNameCaller
                                   )
                , szModNameCallee
                , dwOrdinal
                ) ;
      SetLastError( ERROR_PROC_NOT_FOUND) ;
      return 0 ;                                // error return
    }  // if pProcName is not a string pointer
    // import by name
    // search the target's secure export tables
    DWORD dwHashVal = HashString( pProcName) ;
    if ( dwHashVal <= MAX_SECURE_ORDINAL)  {
      LeaveCriticalSection( &pValModCallee->cs) ;
      char szModNameCaller[ MAX_PATH] ;
      LogSecError( "%u %s %s %s %u"
                , ERROR_SECURE_DYN_LINK_HASH
```

```
              , GetCallerFileName( * ( PDWORD( &hmodTarget) -1)
                                 , szModNameCaller
                                 )
              , szModNameCallee
              , pProcName
              , dwHashVal
              ) ;
    SetLastError( ERROR_PROC_NOT_FOUND) ;
    return 0 ;                                  // error return
  }  // if the name hashed to an embargoed ordinal
  pfnResult = GetProcAddressExp( pValModCallee, dwHashVal, DWORD( -1)) ;
  LeaveCriticalSection( &pValModCallee->cs) ;
  if ( pfnResult != 0)                          // if successful
    return pfnResult ;                          // return the function point
er
  char szModNameCaller[ MAX_PATH] ;
  LogSecError( "%u %s %s %s"
             , ERROR_SECURE_DYN_LINK_NAME
             , GetCallerFileName( * ( PDWORD( &hmodTarget) -1)
                                , szModNameCaller
                                )
             , szModNameCallee
             , pProcName
             ) ;
  SetLastError( ERROR_PROC_NOT_FOUND) ;
  return 0 ;                                    // error return
}  // SecureGetProcAddress
// All functions below are private to this file
/* function CompareDWORD:
   Comparison function used in our bsearch call
*/
static int __cdecl CompareDWORD( const void *e1, const void *e2)
{
  DWORD dw1 = * PDWORD( e1)
      , dw2 = * PDWORD( e2)
      ;
  if ( dw1 < dw2) return -1 ;
  if ( dw1 > dw2) return  1 ;
  return 0 ;
}  // CompareDWORD
/* function FindTableSlot:
   Find a matching slot in the table that tracks validated modules
   Several error conditions fault rather than return to the caller.
   Critical section csTable is held only inside this function, and is
    not held during any GetModuleXXX calls.
*/
static PVALIDATEDMODULE FindTableSlot( HMODULE hmod, bool fAddIfNotFound
)
{
  // convert 0 to the handle of the process' EXE
  if ( hmod == 0)  {
    hmod = GetModuleHandle( 0) ;
    if ( hmod == 0)
      LogSecFatal( "%u %d", ERROR_SECURE_MODULE_HANDLE_1, GetLastError()
) ;
  }  // if the caller wants the current EXE
  // find the file name
  char szFileName[ MAX_PATH] ;
  DWORD dwSizeName = GetModuleFileName( hmod, szFileName, MAX_PATH) ;
  if ( dwSizeName == 0)  {
    LogSecFatal( "%u %X %d"
               , ERROR_SECURE_MODULE_HANDLE_2
               , hmod
               , GetLastError()
               ) ;
  }  // if GetModuleFileName failed
  if ( dwSizeName >= MAX_PATH)  {           // if name is too long
    szFileName[ MAX_PATH - 1] = 0 ;         // ensure a null terminator
```

```
    LogSecFatal( "%u %s", ERROR_SECURE_FILE_NAME, szFileName) ;
  }  // if GetModuleFileName failed
  // ensure the caller hasn't doctored Win32's copy of his file name,
  //  in an attempt to trick us
  HMODULE hmodValidate = GetModuleHandle( szFileName) ;
  if ( hmodValidate != hmod)  {
    LogSecFatal( "%u %s"
               , ERROR_SECURE_BAD_MODULE_NAME
               , szFileName
               ) ;
  }  // if the module file name does not lead back to the right handle
  // hash the file name
  DWORD dwHashName = HashString( szFileName) ;
  // serialize access to the table
  CCritSecAutoRelease csAuto( &csTable) ;
  // find the slot to use in the tables
  std::multimap<DWORD,PVALIDATEDMODULE>::iterator iterValMod
    = pmapValMod->find( dwHashName) ;
  if ( iterValMod != pmapValMod->end())  {
    if ( lstrcmpi( iterValMod->second->pszName, szFileName) == 0)
      return iterValMod->second ;
    for ( iterValMod++
        ; iterValMod != pmapValMod->end()
          &&  iterValMod->second->dwHash == dwHashName
        ; iterValMod++
        )  {
      if ( lstrcmpi( iterValMod->second->pszName, szFileName) == 0)
        return iterValMod->second ;
    }  // check for hash synonyms, very unlikely
  }  // if this hash is mapped
  // return empty handed if we weren't asked to add a new entry
  if ( fAddIfNotFound == false)
    return 0 ;
  // ensure there is a free slot
  if ( nbrModules >= MAX_MODULES)
    LogSecFatal( "%u %s", ERROR_SECURE_VALIDATION_TABLE_FULL, szFileName
) ;
  // allocate the new slot
  // The allocated string is never freed during the life of this process
.
  PVALIDATEDMODULE pValModNew = pValModBase + nbrModules++ ;
  const DWORD dwSize = lstrlen( szFileName) + 1 ;
  pValModNew->pszName = new char[ dwSize] ;
  if ( pValModNew->pszName == 0)
    LogSecFatal( "%u %u", ERROR_SECURE_NOT_ENOUGH_MEMORY_3, dwSize) ;
  lstrcpy( pValModNew->pszName, szFileName) ;
  pValModNew->dwHash = dwHashName ;
  pmapValMod->insert( std::make_pair( dwHashName, pValModNew)) ;
  // save the block pointer for this DLL
  // This will let us skip the lookup and its critical section request
  //  in the SUVW path, which may be useful.
  if ( g_hModThis == hmod)
    pValModThis = pValModNew ;
  // return the slot found
  return pValModNew ;
}  // FindTableSlot
/* function FixupStaticLinkFunnel:
   Back patch a funnel so that it calls the desired code correctly
    and never again reaches here
   After we back patch, we adjust the stack and reexecute the patched
    instructions.
   Open: Validate the instructions we are patching to avoid mysterious
          failure modes.  If we validate too carefully, we will lose
          the ability to recover from a partially applied patch.
         Fail cleanly for all errors.
*/
static FARPROC WINAPI FixupStaticLinkFunnel( PVALIDATEDMODULE pValModNot
Yet
```

```
                                          , PIMPORTTBL pTable
                                          )
{
  // retrieve forbidden fruit from the stack
  DWORD dwAddrFunnelRet = * ( PDWORD( &pValModNotYet) - 1) ;// ret in th
e funnel
  // find the caller's VALIDATEDMODULE block
  // Since modules with secure exports must self-validate, this block
  //  must exist.
  // The pValModNotYet pointer contains bogus data, since we have not
  //  yet patched the caller.
  PVALIDATEDMODULE pValModCaller
    = ValidatedModuleFromCodeAddress( PVOID( dwAddrFunnelRet)) ;
  if ( pValModCaller == 0)                  // if no VALIDATEDMODULE blo
ck
    return 0 ;                              // bad things will happen
  // there is no harm if two threads race through this code, since
  //  they do the same things to the same addresses (except, of course,
  //  to the stack return address, which is unique in each thread)
  // change the funnel to push the VALIDATEDMODULE block
  PatchDWord( dwAddrFunnelRet - 9, DWORD( pValModCaller)) ;
  // change the funnel to call ResolveStaticLink instead of here
  PatchDWord( dwAddrFunnelRet - 4
            , DWORD( ResolveStaticLink) - dwAddrFunnelRet
            ) ;
  // change the return address in the stack to rerun the patched
  //  instructions
  * ( PDWORD( &pValModNotYet) - 1) = dwAddrFunnelRet - 15 ;
  // return to the chosen point
  // The return code is ignored.
  return 0 ;
}  // FixupStaticLinkFunnel
/* function GetCallerFileName:
   Derive the module file name from the caller's return address
   We return a pointer to the result, for his convenience.  In case of
    error, the result may be a constant string.
   We assume the size of the caller's buffer is least MAX_PATH.
*/
static const char * GetCallerFileName( DWORD dwRetAddr
                                     , char * pszModNameCaller
                                     )
{
  HMODULE hmodCaller = HModuleFromCodeAddress( PVOID( dwRetAddr)) ;
  if ( hmodCaller == 0)                     // if bogus return address
    return szUnknown ;                      // error return
  DWORD dwSize = GetModuleFileName( hmodCaller, pszModNameCaller, MAX_PA
TH) ;
  if ( dwSize >= MAX_PATH)                  // if file name is too big
    return szUnknown ;                      // error return
  return pszModNameCaller ;
}  // GetCallerFileName
/* function GetProcAddressExp:
   Return a function's address after validating the caller
   We assume the caller holds this block's critical section.
   Open:  Should this function be in a class to reduce the pValMod->
           usage?
*/
static FARPROC GetProcAddressExp( PVALIDATEDMODULE pValMod
                                , DWORD dwHash
                                , DWORD dwHint
                                )
{
  // ensure the pValMod pointer is still valid
  if ( IsModulePatched( pValMod) == false)  {
    LogSecError( "%u %s %X %d"
               , ERROR_SECURE_BAD_LINK_1
               , pValMod->pszName
               , dwHash
```

```
                , dwHint
                ) ;
    SetLastError( ERROR_MOD_NOT_FOUND) ;
    return 0 ;                                  // return empty handed
  }
  // first try the hint, then do a binary search if necessary
  // The hint is an impossible value (-1) for a dynamic link.
  DWORD ndxExport ;
  if ( dwHint < pValMod->nbrExports
       &&  pValMod->pdwHashExports[ dwHint] == dwHash)  {
    ndxExport = dwHint ;
  }
  else  {                                       // if the hint did not help
    PVOID pvFound = bsearch( &dwHash
                           , pValMod->pdwHashExports
                           , pValMod->nbrExports
                           , sizeof( DWORD)
                           , CompareDWORD
                           ) ;
    if ( pvFound == 0)  {
      LogSecError( "%u %s %X %d"
                 , ERROR_SECURE_BAD_LINK_2
                 , pValMod->pszName
                 , dwHash
                 , dwHint
                 ) ;
      SetLastError( ERROR_PROC_NOT_FOUND) ;
      return 0 ;                                // return empty handed
    }
    ndxExport = PDWORD( pvFound) - pValMod->pdwHashExports ;
  }  // if the hint did not help
#if 0                                           // testing only
  // trace all resolved links
  LogSecError( "%u %s %X %d %X + %X"
             , INFO_SECURE_RESOLVED_LINK
             , pValMod->pszName
             , dwHash
             , dwHint
             , pValMod->dwModBaseAddr
             , pValMod->pdwRVAExports[ ndxExport]
             ) ;
#endif
  return FARPROC( pValMod->dwModBaseAddr + pValMod->pdwRVAExports[ ndxEx
port]) ;
}  // GetProcAddressExp
/* function GetSizeOfPushInstruction:
   Validate the format of a PUSH instruction, and return its size in byt
es
   The size is 2 for a short push, 5 for a long push.
   The function does not return if the instruction is invalid.
*/
static int GetSizeOfPushInstruction( DWORD dwPushed
                                   , DWORD dwAddrAfterPush
                                   , PVALIDATEDMODULE pValMod
                                   )
{
  // Note that an alias cannot occur, since it would be of the form
  //   68.xx.yy.6A.zz, where zz.6A.yy.xx, cast as a signed value, was
  //   between -128 and 127.
  LONG lPushed = LONG( dwPushed) ;
  // try a short push first
  if ( lPushed >= -128  &&  lPushed <= 127)  {
    const DWORD sizePush = 2 ;
    DWORD dwAddrPush = dwAddrAfterPush - sizePush ;
    PBYTE pbInst = PBYTE( dwAddrPush) ;
    if ( IsBadReadPtr( pbInst, sizePush))  {
      LogSecFatal( "%u %s %X + %X"
                 , ERROR_SECURE_BAD_PUSH_1
```

```
                , pValMod->pszName
                , pValMod->dwModBaseAddr
                , dwAddrPush - pValMod->dwModBaseAddr
                ) ;
    }  // if not a valid short push
    if ( pbInst[ 0] != 0x6a  ||  pbInst[ 1] != BYTE( dwPushed & 0xff))
{
      LogSecFatal( "%u %s %X + %X  %2.2X %2.2X"
                , ERROR_SECURE_BAD_PUSH_2
                , pValMod->pszName
                , pValMod->dwModBaseAddr
                , dwAddrPush - pValMod->dwModBaseAddr
                , pbInst[ 0]
                , pbInst[ 1]
                ) ;
    }  // if not a valid short push
    // return the size to the caller
    return sizePush ;
  }  // if a short push
  // must be a long push
  const DWORD sizePush = 5 ;
  DWORD dwAddrPush = dwAddrAfterPush - sizePush ;
  PBYTE pbInst = PBYTE( dwAddrPush) ;
  if ( IsBadReadPtr( pbInst, sizePush))  {
    LogSecFatal( "%u %s %X + %X"
               , ERROR_SECURE_BAD_PUSH_3
               , pValMod->pszName
               , pValMod->dwModBaseAddr
               , dwAddrPush - pValMod->dwModBaseAddr
               ) ;
  }  // if not a valid long push
  if ( pbInst[ 0] != 0x68  ||  * PDWORD( pbInst + 1) != dwPushed)  {
    LogSecFatal( "%u %s %X + %X  %2.2X %2.2X %2.2X %2.2X %2.2X"
               , ERROR_SECURE_BAD_PUSH_4
               , pValMod->pszName
               , pValMod->dwModBaseAddr
               , dwAddrPush - pValMod->dwModBaseAddr
               , pbInst[ 0], pbInst[ 1], pbInst[ 2], pbInst[ 3]
               , pbInst[ 4]
               ) ;
  }  // if not a valid long push
  // return the size to the caller
  return sizePush ;
}  // GetSizeOfPushInstruction
/* function HashString:
   Hash a null-terminated string to a DWORD
   We currently use CRC-32.  If we change the algorithm, we must make
    the same change to the hasher in SecurePE.exe, and modules using
    the old hasher will not work with those using the new hasher.
   The caller is responsible for ensuring all bytes of the string,
    including the null terminator, are accessible.
*/
static DWORD HashString( LPCSTR pszString)
{
  // hash the function name
  DWORD dwHash = 0xffffffff ;               // initial value
  PBYTE pb ;                                // byte stream input
  for ( pb = PBYTE( pszString) ; *pb != 0 ; pb++)  {
    dwHash = ( ( dwHash >> 8) & 0x00ffffff)
           ^ pCRCTable[ ( dwHash ^ *pb) & 0xff] ;
  }
  return dwHash ^ 0xffffffff ;              // final transformation
}  // HashString
/* function IsModulePatched:
   Has a self-validated module been patched?
   Among the uses of this function is to check if a validated module
    has been unloaded and reloaded, whether at the same address or a
    different one.
```

```
   We assume the caller holds this block's critical section.
*/
static bool IsModulePatched( PVALIDATEDMODULE pValMod)
{
  // exit early if we haven't even stuffed our table yet
  if ( pValMod->dwAddrSecureStaticLink == 0)
    return false ;
  // guard all the code, since we don't know what we'll find if our
  //  information is stale
  __try  {                                    // __try / __except
    // ensure the code addresses we are about to inspect are readable
    // This test is technically superfluous, since the SEH block
    //  will properly handle stale values.  We added the test on
    //  24Dec2003 for Scott, Sky and the AV team, since they have
    //  configured the debugger to break on all first chance
    //  exceptions, and this exception got in their way.
    // The test is not airtight, but it should catch most real world
    //  cases when the old patch target is no longer addressable.
    if ( HModuleFromCodeAddress( PVOID( pValMod->dwAddrSecureStaticLink
+ 4))
         != HMODULE( pValMod->dwModBaseAddr))  {
      return false ;
    }
    // check for a near JMP
    if ( * PBYTE( pValMod->dwAddrSecureStaticLink) != 0xe9)
      return false ;                          // not what we expected
    // check for the right relative jump offset
    DWORD dwExpected = DWORD( &FixupStaticLinkFunnel)
                     - ( pValMod->dwAddrSecureStaticLink + 5)
                     ;
    if ( * PDWORD( pValMod->dwAddrSecureStaticLink + 1) != dwExpected)
      return false ;                          // not what we expected
  }  // __try
  __except ( EXCEPTION_EXECUTE_HANDLER)  {
    // our pointers must be bogus
    // Reset some of the stale information.
    pValMod->dwModBaseAddr = 0 ;
    pValMod->dwAddrSecureStaticLink = 0 ;
    pValMod->noPatch.clear() ;
    if ( pValMod->pDynLink != 0)  {
      psetDynLink->erase( pValMod->pDynLink) ;// stop tracking it
      delete pValMod->pDynLink ;
      pValMod->pDynLink = 0 ;
    }  // if there is a dynamic link hook
    // return an error to the caller
    return false ;
  }  // __except
  // successful return
  return true ;
}  // IsModulePatched
/* function MyIsBadStringPointer:
   Validate a string pointer to avoid faults when the OS is not
    sufficiently defensive
   In Win2K, IsBadStringPtr tests only the first and last bytes.
    For a string shorter than a full page (4096 bytes), this test
    is almost good enough.  It still does not handle the case of a
    long run of nonzero characters.
   For a faster technique, we could read every 4096th byte, then stuff
    a 0 in the last byte in the typical case when none of the bytes
    tested is 0.  But the 0 stuff may unintentionally corrupt other
    data.
*/
static bool MyIsBadStringPointer( LPCTSTR lpsz, UINT_PTR ucchMax)
{
  __try  {
    if ( memchr( lpsz, 0, ucchMax) != 0)     // if null terminator found
      return false ;                          // the string is good
  }  // __try
```

```
  __except ( EXCEPTION_EXECUTE_HANDLER)  {
  }  // __except
  // either we did not find a null terminator soon enough, or we faulted
  //  on a memory access
  return true ;                                 // the string is bad
}  // MyIsBadStringPointer
/* function newDYNLINK:
   Construct a new object
   This silly function avoids compiler error C2712 when we compile with
    the -GX option.
*/
static DYNLINK * newDYNLINK()
{
  return new DYNLINK ;
}  // newDYNLINK
/* function ParseSecureExports:
   Parse the secure export table
   This function is called after a successful self-validation.
*/
static bool ParseSecureExports( PVALIDATEDMODULE pValMod)
{
  // point to the module base
  PBYTE pbMod = PBYTE( pValMod->dwModBaseAddr) ;
  // parse the secure export directory
  // Relocate the function RVAs and save pointers to the table.
  bool fResult = false ;                        // assume failure
  __try  {                                      // __try / __except
    // find the export directory in this module file
    PIMAGE_DOS_HEADER pHdrDOS = PIMAGE_DOS_HEADER( pbMod) ;
    if ( pHdrDOS->e_magic != 'ZM')
      return false ;                            // error return
    PIMAGE_NT_HEADERS32 pHdrNT
      = PIMAGE_NT_HEADERS32( pbMod + pHdrDOS->e_lfanew) ;
    if ( pHdrNT->Signature != 'EP')
      return false ;                            // error return
    if ( pHdrNT->FileHeader.Machine != IMAGE_FILE_MACHINE_I386)
      return false ;                            // error return
    if ( ( pHdrNT->FileHeader.Characteristics & IMAGE_FILE_EXECUTABLE_IM
AGE)
         == 0)  {
      return false ;                            // error return
    }
    if ( pHdrNT->OptionalHeader.Magic != IMAGE_NT_OPTIONAL_HDR32_MAGIC)
      return false ;                            // error return
    PIMAGE_DATA_DIRECTORY pDirExport
      = pHdrNT->OptionalHeader.DataDirectory + IMAGE_DIRECTORY_ENTRY_EXP
ORT ;
    // does this module have an export directory?
    // ZAPro.exe and IClient.exe do not.
    if ( pDirExport->VirtualAddress == 0  ||  pDirExport->Size == 0)
      return false ;
    // the secure export directory immediately follows the regular one
    // Open:  Should we add a validation field to the header?
    PDWORD pdwNbrExports = PDWORD( pbMod
                                   + pDirExport->VirtualAddress
                                   + pDirExport->Size
                                 ) ;
    pValMod->nbrExports = *pdwNbrExports ;
    PDWORD pdwSignature = pdwNbrExports + 1 + pValMod->nbrExports * 2 ;
    if ( *pdwSignature != ( pValMod->nbrExports ^ 0x5aa5a55a) * dwLargeP
rime)
      return false ;                            // if the signature is not v
alid
    pValMod->pdwRVAExports = pdwNbrExports + 1 ;
    pValMod->pdwHashExports = pValMod->pdwRVAExports + pValMod->nbrExpor
ts ;
    // successful return
    fResult = true ;
```

```
  }  // __try
  __except ( EXCEPTION_EXECUTE_HANDLER) {
  }  // __except
  // return to the caller
  return fResult ;
}  // ParseSecureExports
/* function SecureLoadLibrary:
   Load a module with secure exports, and return its PVALIDATEDMODULE
    block
*/
static PVALIDATEDMODULE WINAPI SecureLoadLibrary( LPCSTR lpFileName)
{
  // load the module
  // Since we usually have an unqualified file name, we don't know the
  //  full path name until after the load succeeds.
  HMODULE hmod = LoadLibrary( lpFileName) ;
  if ( hmod == 0)
    return 0 ;
  // find the PVALIDATEDMODULE block
  // Since modules with secure exports must self-validate, this block
  //  must exist.
  return FindTableSlot( hmod, false) ;
}  // SecureLoadLibrary
/* function SetTheGetProcAddressHook:
   Hook the address in the caller's import table
   We find KERNEL32.dll in the import directory, then GetProcAddress
    in the import lookup table, then the desired address at the matching
    offset in the import address table.  This technique supports a prior
    GetProcAddress hook by another program.
   Since we support prior hooking of GetProcAddress, we need a place to
    save the prior address.  So we allocate memory.  But then the hook
    needs to find that memory.  So we provide it as an argument to the
    hook.  We put the code that does this in the allocated memory.
   The problem with allocating is that we can leak process memory.  We
    associate memory with a DLL, but we don't know when that DLL is
    unloaded.  If the EXE repeatedly bounces the DLL, we will accumulate
    memory for the DLL.  We could map the memory by the module name, as
    we do for the VALIDATEDMODULE control blocks, but the application
    can trick us by loading each new copy with a temporary file name, as
    the Wise Installer does.  Also, using a table structure, or links
    among control blocks, requires serialization calls to avoid
    threading problems.  We may try hooking the DLL's DLL_PROCESS_
    DETACH, but then we must somehow guarantee that VSInit.dll is not
    unloaded first.
   One advantage of allocating is that this creates an opportunity
    in the future to add a validation field, to make it harder for
    an attacker to gain access to all our secured dynamic links.
   Open:  For modules that have just been through ParseSecureExports,
           the validation here is redundant.
          Consider adding unique log calls in each error path.
*/
static PDYNLINK SetTheGetProcAddressHook( HMODULE hmod)
{
  // get the module file name early, so that we can use a private
  //  critical section later without worrying about deadlock
  // See the file prolog for more information about deadlock risks.
  char szFileName[ MAX_PATH] ;
  DWORD dwSizeName = GetModuleFileName( hmod, szFileName, MAX_PATH) ;
  if ( dwSizeName == 0)  {
    LogSecFatal( "%u %X %d"
               , ERROR_SECURE_MODULE_HANDLE_3
               , hmod
               , GetLastError()
               ) ;
  }  // if GetModuleFileName failed
  if ( dwSizeName >= MAX_PATH)  {           // if name is too long
    szFileName[ MAX_PATH - 1] = 0 ;         // ensure a null terminator
    LogSecFatal( "%u %s", ERROR_DYNLINK_FILE_NAME, szFileName) ;
```

```
  }  // if GetModuleFileName failed
  // we protect this access in case we have a bogus module handle
  // The success of the call above makes a bogus module handle very
  //  unlikely.
  PDWORD pdwIAT ;                               // import address table
  PFNGETPROCADDRESS pGetProcAddress ;           // pointer before we patch
  int ndxGetProcAddress ;                       // index in import lookup ta
ble
  try  {                                        // try / catch
    // point to the module base
    PBYTE pbMod = PBYTE( hmod) ;
    // find the import directory in this module file
    PIMAGE_DOS_HEADER pHdrDOS = PIMAGE_DOS_HEADER( pbMod) ;
    if ( pHdrDOS->e_magic != 'ZM')
      return 0 ;                                // error return
    PIMAGE_NT_HEADERS32 pHdrNT
      = PIMAGE_NT_HEADERS32( pbMod + pHdrDOS->e_lfanew) ;
    if ( pHdrNT->Signature != 'EP')
      return 0 ;                                // error return
    if ( pHdrNT->FileHeader.Machine != IMAGE_FILE_MACHINE_I386)
      return 0 ;                                // error return
    if ( ( pHdrNT->FileHeader.Characteristics & IMAGE_FILE_EXECUTABLE_IM
AGE)
         == 0)  {
      return 0 ;                                // error return
    }
    if ( pHdrNT->OptionalHeader.Magic != IMAGE_NT_OPTIONAL_HDR32_MAGIC)
      return 0 ;                                // error return
    PIMAGE_DATA_DIRECTORY pDirImport
      = pHdrNT->OptionalHeader.DataDirectory + IMAGE_DIRECTORY_ENTRY_IMP
ORT ;
    // does this module have an import directory?
    // Since every module of interest does, we probably can remove this
    //  test.
    if ( pDirImport->VirtualAddress == 0  ||  pDirImport->Size == 0)
      return 0 ;
    // find the import descriptor for KERNEL32.dll
    // We could consider using pDirImport->Size to bound the loop, but
    //  the documentation claims the array is terminated by an entry
    //  of all zeroes.
    PIMAGE_IMPORT_DESCRIPTOR pImportDescriptor ;
    for ( pImportDescriptor
          = PIMAGE_IMPORT_DESCRIPTOR( pbMod + pDirImport->VirtualAddress
)
        ; pImportDescriptor->Name != 0
          &&  lstrcmpi( PSTR( pbMod + pImportDescriptor->Name), szKernel
32) != 0
        ; pImportDescriptor++
        )  {
    }  // loop until we find the KERNEL32.dll import descriptor
    if ( pImportDescriptor->Name == 0)      // if KERNEL32.dll not found
      return 0 ;                                // error return
    // search the KERNEL32 import lookup table for GetProcAddress
    PDWORD pdwLookup ;
    for ( pdwLookup = PDWORD( pbMod + pImportDescriptor->OriginalFirstTh
unk)
          , ndxGetProcAddress = 0
        ; *pdwLookup != 0
        ; pdwLookup++
          , ndxGetProcAddress++
        )  {
      // we expect an import by name
      if ( ( *pdwLookup & IMAGE_ORDINAL_FLAG32) != 0)
        continue ;                              // if import by ordinal
      // compare the name, which comes after the hint
      if ( lstrcmp( PSTR( pbMod + *pdwLookup + sizeof( WORD))
                  , szGetProcAddress
                  ) == 0)  {
```

```
        break ;
      }  // if the desired function
    }  // loop until we find the GetProcAddress slot
    if ( *pdwLookup == 0)                    // if GetProcAddress not fou
nd
      return 0 ;                             // error return
    // save the address of the caller's GetProcAddress
    pdwIAT = PDWORD( pbMod + pImportDescriptor->FirstThunk) ;
    pGetProcAddress = PFNGETPROCADDRESS( pdwIAT[ ndxGetProcAddress]) ;
    // is this module's GetProcAddress call already hooked by us?
    // This call, for a secured export in VSInit, succeeds only if
    //  the module's GetProcAddress call is already hooked.
    // Do not hold any private critical sections during this call,
    //  since the call will reach FindTableSlot if the module's
    //  GetProcAddress is already hooked, and that function calls
    //  GetModuleFileName.
    if ( pGetProcAddress( g_hModThis, szDummyImport))
      return 0 ;
  }  // try
  catch (...) {
    return 0 ;
  }  // catch
  // now serialize access to avoid a race between two threads to
  //  hook the same DLL
  // Since we don't have a control block just for this module we use
  //  a critical section for the entire process.
  // We can't use our CCritSecAutoRelease class here without breaking
  //  up this function, since MSVC does not allow SEH to coexist with
  //  objects requiring destruction (warning C4509).
  EnterCriticalSection( &csDynLinkHook) ;
  // bail out if another thread has just set the hook
  // We ignore the case where a program other than ours races to
  //  set the hook.  We blithely assume that is a very rare event.
  if ( pGetProcAddress != PFNGETPROCADDRESS( pdwIAT[ ndxGetProcAddress])
)  {
    LeaveCriticalSection( &csDynLinkHook) ;
    return 0 ;                               // bail out
  }
  // allocate a block for this hook
  PDYNLINK pDynLink = newDYNLINK() ;
  pDynLink->pGetProcAddress = pGetProcAddress ;
  const DWORD dwSize = dwSizeName + 1 ;
  pDynLink->pszName = new char[ dwSize] ;
  if ( pDynLink->pszName == 0)
    LogSecFatal( "%u %u", ERROR_SECURE_NOT_ENOUGH_MEMORY_3, dwSize) ;
  lstrcpy( pDynLink->pszName, szFileName) ;
  pDynLink->hmod = hmod ;                    // module handle
  // finish building our hook
  // The hook pops the return address, pushes the address of the
  //  DYNLINK block, pushes the return address back, then jumps to
  //  the common hook.
  * PDWORD( pDynLink->bHookGetProcAddress + 2) = DWORD( pDynLink) ;
  * PDWORD( pDynLink->bHookGetProcAddress + 8)
    = DWORD( ResolveDynamicLink) - DWORD( pDynLink->bHookGetProcAddress
+ 12) ;
  // point the IAT to our hook
  pDynLink->pdwPatchTarget = pdwIAT + ndxGetProcAddress ;
  PatchDWord( DWORD( pDynLink->pdwPatchTarget)
            , DWORD( pDynLink->bHookGetProcAddress)
            ) ;
  // insert the block into the tracking set
  psetDynLink->insert( pDynLink) ;
  // return to the caller
  LeaveCriticalSection( &csDynLinkHook) ;
  return pDynLink ;
}  // SetTheGetProcAddressHook
/* function TerminateSecureAPI:
   Destructor
```

```
   Free the constructor's allocations, to avoid memory leaks.
*/
static void __cdecl TerminateSecureAPI()
{
  // validated modules
  // We delete these before the dynamic link hooks, since these
  //  destructors delete some elements of psetDynLink.
  if ( pmapValMod != 0)  {
    // free the dynamic allocations in each block
    std::multimap<DWORD,PVALIDATEDMODULE>::iterator iterValMod ;
    for ( iterValMod = pmapValMod->begin()
        ; iterValMod != pmapValMod->end()
        ; iterValMod++
        )  {
      if ( iterValMod->second->pszName != 0)// if there is a file name
        delete [] iterValMod->second->pszName ;// delete the name
      if ( iterValMod->second->pDynLink != 0)  {
        psetDynLink->erase( iterValMod->second->pDynLink) ;// stop track
ing it
        delete iterValMod->second->pDynLink ;
        iterValMod->second->pDynLink = 0 ;
      }  // if there is a dynamic link hook
    }  // loop once for each dynamic link hook
    // empty the map
    pmapValMod->clear() ;
    // destroy the map
    delete pmapValMod ;
  }  // if pmapValMod was allocated
  // dynamic link hooks
  if ( psetDynLink != 0)  {
    // loop until the dynamic link set is empty
    std::set<PDYNLINK>::iterator iterDynLink ;
    for ( iterDynLink = psetDynLink->begin()
        ; iterDynLink != psetDynLink->end()
        ; iterDynLink = psetDynLink->begin()
        )  {
      PDYNLINK pDynLinkToDelete = *iterDynLink ;
      UndoTheGetProcAddressHook( pDynLinkToDelete) ;// unhook GetProcAdd
ress
      psetDynLink->erase( iterDynLink) ;    // remove from the set
      delete pDynLinkToDelete ;             // delete the block
    }  // loop once for each dynamic link hook
    // destroy the set
    delete psetDynLink ;
  }  // if psetDynLink was allocated
  // destroy the critical sections
  DeleteCriticalSection( &csTable) ;
  DeleteCriticalSection( &csDynLinkHook) ;
  // free the CRC table
  if ( pCRCTable != 0)
    HeapFree( GetProcessHeap(), 0, pCRCTable) ;
}  // TerminateSecureAPI
/* function UndoTheGetProcAddressHook:
   Restore the address in the caller's import table
*/
static bool UndoTheGetProcAddressHook( const DYNLINK * pDynLink)
{
  // validate the module handle and file name
  char szFileName[ MAX_PATH] ;
  DWORD dwSizeName = GetModuleFileName( pDynLink->hmod, szFileName, MAX_
PATH) ;
  if ( dwSizeName == 0  ||  dwSizeName >= MAX_PATH)
    return false ;
  if ( lstrcmpi( szFileName, pDynLink->pszName) != 0)
    return false ;
  // validate the current import table contents
  // If someone else hooked the value after us, we do not restore the
  //  original value.  If that other hook calls us after we unload, the
```

```
  //  application will fault.
  HMODULE hmodValidate = HModuleFromCodeAddress( pDynLink->pdwPatchTarge
t) ;
  if ( hmodValidate != pDynLink->hmod)
    return false ;
  if ( * ( pDynLink->pdwPatchTarget) != DWORD( pDynLink->bHookGetProcAdd
ress))
    return false ;
  // restore the original address
  PatchDWord( DWORD( pDynLink->pdwPatchTarget)
            , DWORD( pDynLink->pGetProcAddress)
            ) ;
  // return to the caller
  return true ;
}  // UndoTheGetProcAddressHook
/* function ValidatedModuleFromCodeAddress:
   Derive a control block pointer
   Open:  Validate the control block, including whether the caller's
          code address is within the .text section.
*/
static PVALIDATEDMODULE ValidatedModuleFromCodeAddress( PVOID pvAddr)
{
  // find the caller's module handle
  HMODULE hmod = HModuleFromCodeAddress( pvAddr) ;
  if ( hmod == 0)                              // if no module handle
    return 0 ;                                 // error return
  // find the caller's VALIDATEDMODULE block
  return FindTableSlot( hmod, false) ;
}  // ValidatedModuleFromCodeAddress
/* function IfValidationSucceeded:
   Validate the module that raised the exception
*/
bool IfValidationSucceeded( PVALIDATESELF pvalSelf)
{
    // get the module handle
    HMODULE hmod = HModuleFromCodeAddress( pvalSelf->EIPInCaller );
    if( hmod == 0 )
        return false;
    // validate the requested module
    // The call returns only if validation succeeds.
    return IsModuleValid( hmod, pvalSelf );
}  // IfValidationSucceeded
// SecureError.cpp
// Copyright (c) 2004. Zone Labs, LLC All Rights Reserved.
/* File SecureError.cpp:
   Report errors related to module validation or secure API
*/
// Pre-compiled header files, must come first
#include "VSInit_pch.h"
#pragma hdrstop
// Compiler header files
#include <tchar.h>
#include <stdio.h>
#include <stdarg.h>
// Application header files
#include <vsinit.h>
#include "vsinit_int.h"
// Constants
const char szPfx[] = "[SAPI] "               // stands for Secure API
         , szSfx[] = "\n"                    // terminate the line
         ;
// Local data
static DWORD ndxTLS = TLS_OUT_OF_INDEXES ;  // index of thread local sto
rage
// Local functions
static void LogSecCommon( va_list * pmarker, const char * fmt, DWORD dwL
ogFlag ) ;
static bool MyIsBadStringPointer( LPCTSTR lpsz, UINT_PTR ucchMax) ;
```

```
static void __cdecl TerminateSecureErrorLogging() ;
/* function EnableSecureErrorLogging:
   Set logging on or off for the current thread
*/
void EnableSecureErrorLogging( DWORD dwLogFlag)
{
  // set the value
  if ( ndxTLS != TLS_OUT_OF_INDEXES)
    TlsSetValue( ndxTLS, PVOID( dwLogFlag)) ;
}  // EnableSecureErrorLogging
/* function InitializeSecureErrorLogging:
   Allocate resources
*/
bool InitializeSecureErrorLogging()
{
  // specify the termination function
  atexit( TerminateSecureErrorLogging) ;
  // allocate the thread local storage index
  ndxTLS = TlsAlloc() ;
  if ( ndxTLS == TLS_OUT_OF_INDEXES)
    return false ;
  // the value is initially true
  TlsSetValue( ndxTLS, PVOID( true)) ;
  // successful return
  return true ;
}  // InitializeSecureErrorLogging
/* function LogSecError:
   Write error information
*/
void __cdecl LogSecError( const char * fmt, ...)
{
  // exit early if logging is disabled
  DWORD dwLogFlag = SAPI_LOG_TVDEBUG;
  if ( ndxTLS != TLS_OUT_OF_INDEXES
       &&  (dwLogFlag = (DWORD)TlsGetValue( ndxTLS)) == SAPI_LOG_NONE
       &&  GetLastError() == NO_ERROR)  {
    return ;
  }  // if error logging is disabled
  // prepare the argument list
  va_list marker ;
  va_start( marker, fmt) ;
  // do the common work
  LogSecCommon( &marker, fmt, dwLogFlag) ;
}  // LogSecError
/* function LogSecFatal:
   Write error information and die
*/
__declspec( noreturn)
void __cdecl LogSecFatal( const char * fmt, ...)
{
  // prepare the argument list
  va_list marker ;
  va_start( marker, fmt) ;
  // do the common work
  LogSecCommon( &marker, fmt, SAPI_LOG_EVENTLOG) ;
  DbgPrintfEx( ODF_ALWAYS | ODF_ODS | ODF_STACKTRACE, "%sFatal error\n",
 szPfx) ;
  unsigned int dumpFlags = GetEnvironmentInt( "tvdumpflags", 0) ;
  if ( ( dumpFlags & TVDUMP_SECOND_CHANCE) != 0)  {
 char zLogDir[ MAX_PATH] ;
 GetInternetLogsDirectoryEx( zLogDir, sizeof zLogDir) ;
 // this isn't a second chance exception, but we want to go through
 //   that path's impersonation restore
    WriteMemoryDump( dump_second_chance
                  , GetCurrentProcessId()
       , ( dumpFlags & TVDUMP_FULL) ? FULL_DUMP : SMALL_DUMP
       , NULL
       , zLogDir
```

```
      ) ;
  }  // if a dump was requested
  // die a horrible death
  // ExitProcess is rarely used in our code.  We may need to replace it
  //  with something more suitable.
  ExitProcess( 1) ;
}  // LogSecFatal
// All functions below are private to this file
/* function LogSecCommon:
   Processing common to both public functions
*/
static void LogSecCommon( va_list * pmarker, const char * fmt, DWORD dwL
ogFlag )
{
  // limit the buffer
  const int bufSize = 1024
          , lenPfx = sizeof szPfx - sizeof szPfx[ 0]
          , lenSfx = sizeof szSfx - sizeof szSfx[ 0]
          , bufLeft = bufSize - lenPfx - lenSfx
          ;
  char szBuf[ bufSize] ;
  // write the prefix
  memcpy( szBuf, szPfx, lenPfx) ;
  char * pszBuf = szBuf + lenPfx ;
  // guard against a sloppy caller
  // The caller should validate the string pointers he sends us,
  //  since we cannot easily do this.
  const UINT_PTR MAX_FMT_STRING = 1024 ;
  if ( MyIsBadStringPointer( fmt, MAX_FMT_STRING))  {
    LogSecError( "%u", ERROR_SECURE_BAD_LOG_FORMAT) ;
    return ;
  }
  // write the body of the message
  int count = _vsnprintf( pszBuf, bufLeft, fmt, *pmarker) ;
  if ( count == -1 )
    count = _strncnt( pszBuf, bufLeft) ;
  count += lenPfx ;
  // append the suffix
  strcpy( szBuf + count, szSfx) ;
  // log the message
  if ( dwLogFlag == SAPI_LOG_EVENTLOG)
  {
 LPCTSTR pStrings[] = { szBuf };
 VSReportEventID(
   EVENTLOG_ERROR_TYPE,
   EVENT_SECURE_API,
   elementsof(pStrings),
   pStrings );
  }
  else if ( dwLogFlag == SAPI_LOG_TVDEBUG)
 DbgOutput( ODF_ALWAYS|ODF_ODS, count + lenSfx, szBuf) ;
#ifdef _DEBUG
  else
 DbgPrintfEx( ODF_ALWAYS|ODF_ODS|ODF_STACKTRACE, "LogSecCommon invalid l
og flags\n" );
#endif
}  // LogSecCommon
/* function MyIsBadStringPointer:
   Validate a string pointer to avoid faults when the OS is not
    sufficiently defensive
   In Win2K, IsBadStringPtr tests only the first and last bytes.
    For a string shorter than a full page (4096 bytes), this test
    is almost good enough.  It still does not handle the case of a
    long run of nonzero characters.
   For a faster technique, we could read every 4096th byte, then stuff
    a 0 in the last byte in the typical case when none of the bytes
    tested is 0.  But the 0 stuff may unintentionally corrupt other
    data.
```

```
*/
static bool MyIsBadStringPointer( LPCTSTR lpsz, UINT_PTR ucchMax)
{
  __try  {
    if ( memchr( lpsz, 0, ucchMax) != 0)     // if null terminator found
      return false ;                         // the string is good
  }  // __try
  __except ( EXCEPTION_EXECUTE_HANDLER)  {
  }  // __except
  // either we did not find a null terminator soon enough, or we faulted
  //  on a memory access
  return true ;                              // the string is bad
}  // MyIsBadStringPointer
/* function TerminateSecureErrorLogging:
   Release resources
*/
static void __cdecl TerminateSecureErrorLogging()
{
  if ( ndxTLS != TLS_OUT_OF_INDEXES)
    TlsFree( ndxTLS) ;
}  // TerminateSecureErrorLogging
// ValidatePEFile.cpp
// Copyright (c) 2004. Zone Labs, LLC All Rights Reserved.
/* File ValidatePEFile.cpp:
   Validate the code signature and signing organization of a PE file
*/
// Pre-compiled header files, must come first
#include "VSInit_pch.h"
#pragma hdrstop
// Windows header files
#include <wincrypt.h>
#include <wintrust.h>
// Application header files
#include <VSInit.h>
#include "VSInit_int.h"
// Local class to save signature parse information
class CASN1BERNode
{
public:
  CASN1BERNode() ;
  ~CASN1BERNode() ;
  class CASN1BERNode * m_NextSibling ;
  class CASN1BERNode * m_FirstChild ;
  class CASN1BERNode * m_Parent ;
  DWORD                m_Length ;
  DWORD                m_LenHdr ;
  PBYTE                m_RawData ;
  DWORD                m_Tag ;
  BYTE                 m_RawTag ;
  DWORD                m_NbrChildren ;
} ;  // class CASN1BERNode
// Universal types
#define ASN_TYPE_INTEGER            2
#define ASN_TYPE_BITS               3
#define ASN_TYPE_OCTETSTRING        4
#define ASN_TYPE_NULL               5
#define ASN_TYPE_OID                6
#define ASN_TYPE_PRINTABLESTRING   19
#define ASN_TYPE_T61STRING         20
// Macros
#define RoundUp(X, Y) (((X) + (Y) - 1) & ~((Y)-1))// assumes Y = 2^n
#define IsOID(a,b) ( (a)->m_RawTag == ASN_TYPE_OID      \
                    &&  (a)->m_Length == sizeof (b)     \
                    &&  memcmp( (a)->m_RawData, (b), sizeof (b)) == 0)
// Constants
// The compiler emits bloated code for aggregate constants (e.g., arrays
//  or structures) in function scope.
const char * pszValidSigners[] = { "Zone Labs, Inc"
```

```
                                , "AT&T"
                                , "Computer Associates International"
          , "BigPlanet"
          , "NuSkin International"
          , "Check Point Software Technologies Ltd."
                                , "Fiberlink Communications"
          , "Funk Software, Inc."
                                } ;
const int nbrValidSigners = elementsof( pszValidSigners) ;
const WCHAR * pwszValidSigners[] = { L"AT&T"
            , L"Computer Associates International"
                                , L"Fiberlink Communications"
            , L"Funk Software, Inc."
                                } ;
const int nbrValidUniSigners = elementsof( pwszValidSigners) ;
const BYTE bPKCS7SignedData[]               // 1.2.840.113549.1.7.2
            = { 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x01, 0x07, 0x02}
          , bSPCIndirectDataObjID[]         // 1.3.6.1.4.1.311.2.1.4
            = { 0x2B, 0x06, 0x01, 0x04, 0x01, 0x82, 0x37, 0x02, 0x01, 0x0
4}
          , bOID_ORGANIZATION_NAME[]        // 2.5.4.10
            = { 0x55, 0x04, 0x0a}
          ;
// Global data
// Because of this variable, Certificates.cpp's static initializer must
//  run before ours does.
extern HMODULE hmodCrypt32 ;                // does not exist in base OS
R2
// Local functions
static PBYTE ComputeMessageDigest( const BYTE * pbFile) ;
static bool GetSignature( const BYTE * pbMod, CASN1BERNode * pNodeTop) ;
static bool IsThisCertSignedByAFriend( const CASN1BERNode * pNodeCert) ;
static CASN1BERNode * newCASN1BERNode() ;
static LONG OneASN1BERLevel( CASN1BERNode * pNodeParent
                           , const BYTE * pbStart
                           , LONG lBytesLeft
                           ) ;
static void __cdecl TerminateValidatePEFile() ;
/* function InitializeValidatePEFile:
   Static object initialization
   Prepare data structures for module validation.
   This function is called from the carefully sequenced static object
    initialization in VSInit.cpp.
*/
bool InitializeValidatePEFile()
{
  // if OSR2 and IE < 4.0, there is nothing to do
  if ( hmodCrypt32 == 0)
    return true ;
  // specify the termination function
  atexit( TerminateValidatePEFile) ;
  // successful return
  return true ;
}  // InitializeValidatePEFile
/* function IsPEFileValid:
   As it says
   This function is exported.
*/
bool WINAPI IsPEFileValid( const char *pszFileName, DWORD dwLogFlag)
{
 return IsPEFileValidEx(pszFileName, dwLogFlag, PEFV_VALIDATESIGNER) ;
}  // IsPEFileValid
/* function IsPEFileValidEx:
   MSH extension to allow for a WinVerifyTrust alternative
*/
bool IsPEHFileValidEx( HANDLE hFile, DWORD dwLogFlag, DWORD dwOptFlags)
{
  // set error logging flags for this thread
```

```
EnableSecureErrorLogging( dwLogFlag) ;
// variables we may use in the __finally clause
bool fValid = false ;                       // assume the worst
HANDLE hMap = 0;
PVOID pFileData = 0 ;
HCRYPTMSG hCrMsg = 0 ;
CASN1BERNode * pNodeTop = 0 ;
PBYTE pbDigestComputed = 0 ;
HCERTSTORE hStoreMsg = 0 ;
__try  {                                    // __try / __finally
  if ( hFile == INVALID_HANDLE_VALUE)  {
    LogSecError( "%u %d", ERROR_SECURE_OPEN_FILE, GetLastError()) ;
    return false ;                          // error return
  }
  // get the file size
  const LONG ORCA_FILE_SIZE = 1 << 24 ;   // 16 MB
  LONG lSizeFile = GetFileSize( hFile, 0) ;
  if ( lSizeFile <= 0  ||  lSizeFile > ORCA_FILE_SIZE)  {
    LogSecError( "%u %d", ERROR_SECURE_ORCA_FILE, lSizeFile) ;
    return false ;
  }
  // create a mapping
  hMap = CreateFileMapping( hFile
                          , 0             // no security
                          , PAGE_READONLY
                          , 0             // entire file
                          , 0             // entire file
                          , 0             // unnamed
                          ) ;
  if ( hMap == 0)  {
    LogSecError( "%u %d", ERROR_SECURE_MAP_FILE, GetLastError()) ;
    return false ;
  }
  // create a view
  pFileData = MapViewOfFile( hMap
                           , FILE_MAP_READ
                           , 0            // start view at file start
                           , 0            // start view at file start
                           , 0            // view the entire file
                           ) ;
  if ( pFileData == 0)  {
    LogSecError( "%u %d", ERROR_SECURE_VIEWFILEMAP, GetLastError()) ;
    return false ;
  }
  // find the signature
  // We can't allocate pNodeTop on the stack, since that would
  //  conflict with the use of SEH in this function.
  PBYTE pbFileData = PBYTE( pFileData) ;
  pNodeTop = newCASN1BERNode() ;
  if ( pNodeTop == 0)  {
    LogSecError( "%u", ERROR_SECURE_NO_ASN1_NODE) ;
    return false ;
  }
  if ( GetSignature( pbFileData, pNodeTop) == false)  {
    LogSecError( "%u", ERROR_SECURE_NO_SIGNATURE) ;
    return false ;
  }
  // compute the message digest
  pbDigestComputed = ComputeMessageDigest( pbFileData) ;
  if ( pbDigestComputed == 0)  {
    LogSecError( "%u", ERROR_SECURE_COMPUTE_MESSAGE_DIGEST) ;
    return false ;
  }
  const DWORD dwSizeDigest
    = HeapSize( GetProcessHeap(), 0, pbDigestComputed) ;
  // We start with the validation functions we can perform on Win95
  //  OSR2 without IE 4.x or even the Authenticode 2.0 update.
  // find the unencrypted digest and the first certificate
```

```
   CASN1BERNode * pNodeCert
                , * pNodeDgst
                ;
   __try  {                                  // __try / __except
     // find the PKCS-7 signed data
     // Documentation is available at
     //  http://www.rsasecurity.com/rsalabs/pkcs.
     CASN1BERNode * pNodeFind ;
     for ( pNodeFind = pNodeTop->m_FirstChild
         ; pNodeFind != 0  &&  !IsOID( pNodeFind, bPKCS7SignedData)
         ; pNodeFind = pNodeFind->m_NextSibling
         )  {
     }  // loop until we find the PKCS-7 signed data
     // skip some nodes
     pNodeFind = pNodeFind->m_NextSibling ;// data comes after the OID
     do  {
       pNodeFind = pNodeFind->m_FirstChild ;
     } while ( pNodeFind->m_RawTag != ASN_TYPE_INTEGER) ;// version
     pNodeFind = pNodeFind->m_NextSibling ;// digest algorithms
     pNodeFind = pNodeFind->m_NextSibling ;// content information
     // take a detour to find the unencrypted digest
     pNodeDgst = pNodeFind->m_FirstChild ;
     if ( !IsOID( pNodeDgst, bSPCIndirectDataObjID))  {
       LogSecError( "%u", ERROR_SECURE_NO_UNENCRYPTED_DIGEST) ;
       return false ;
     }
     pNodeDgst = pNodeDgst->m_NextSibling ;
     pNodeDgst = pNodeDgst->m_FirstChild ;
     pNodeDgst = pNodeDgst->m_FirstChild ;
     pNodeDgst = pNodeDgst->m_NextSibling ;
     pNodeDgst = pNodeDgst->m_FirstChild ;
     pNodeDgst = pNodeDgst->m_NextSibling ;
     // back to finding the first certificate
     pNodeFind = pNodeFind->m_NextSibling ;// certificates
     pNodeFind = pNodeFind->m_FirstChild ;
     pNodeCert = pNodeFind ;
     if ( pNodeCert == 0)  {
       LogSecError( "%u", ERROR_SECURE_NO_CERTS_IN_SIG) ;
       return false ;
     }
   }  // __try
   __except( DefaultExceptionFilterEx( GetExceptionInformation()))  {
     LogSecError( "%u", ERROR_SECURE_PARSE_CERT_1) ;
     return false ;
   }
   // validate the signer's organization name
   // We look in all certificates.  We assume the Zone Labs certi-
   //  ficate has our company name in the first subject field.
   // OEM clients using the secure API must have a valid signature
   //  that includes a Zone Labs certificate.
   CASN1BERNode * pNodeSign;
if ( dwOptFlags & PEFV_VALIDATESIGNER)
{
 bool fSignedByAFriend ;
 for ( fSignedByAFriend = false
     , pNodeSign = pNodeCert
  ; fSignedByAFriend == false
     &&  pNodeSign != 0
  ; fSignedByAFriend = IsThisCertSignedByAFriend( pNodeSign)
     , pNodeSign = pNodeSign->m_NextSibling
  )  {
 }  // check certificates until we find a Zone signature
 if ( fSignedByAFriend == false)  {
   LogSecError( "%u", ERROR_SECURE_NO_ZONE_SIG) ;
   return false ;
 }
}
else
```

```
  pNodeSign = pNodeCert;
    // validate the unencrypted message digest
    // In Win95 OSR2, dwSizeDigest ranges from 16-20.
    if ( dwSizeDigest < pNodeDgst->m_Length
         ||  memcmp( pbDigestComputed
                   , pNodeDgst->m_RawData
                   , pNodeDgst->m_Length
                   ) != 0)  {
      LogSecError( "%u", ERROR_SECURE_DIGEST_MISMATCH) ;
      return false ;
    }  // if the unencrypted digest does not match
    // this is as far as we can go in OSR2 with IE < 4.0
    if ( hmodCrypt32 == 0)  {
      fValid = true ;
      return true ;
    }
    // create a crypto message object
    hCrMsg = CryptMsgOpenToDecode( X509_ASN_ENCODING | PKCS_7_ASN_ENCODI
NG
                                 , 0        // dwFlags
                                 , 0        // dwMsgType
                                 , 0        // use default crypto provid
er
                                 , 0        // pRecipientInfo
                                 , 0        // pStreamInfo
                                 ) ;
    if ( hCrMsg == 0)  {
      LogSecError( "%u %d", ERROR_SECURE_MSGOPENTODECODE, GetLastError()
) ;
      return false ;
    }
    // add the data to the crypto object
    if ( CryptMsgUpdate( hCrMsg
                       , pNodeTop->m_RawData
                       , pNodeTop->m_Length + pNodeTop->m_LenHdr
                       , TRUE               // final update
                       ) == FALSE)  {
      LogSecError( "%u %d", ERROR_SECURE_CRYPTMSGUPDATE, GetLastError())
 ;
      return false ;
    }
    // validate the signature
    if ( CryptMsgGetAndVerifySigner( hCrMsg, 0, 0, 0, 0, 0) == FALSE)  {
      LogSecError( "%u %d", ERROR_SECURE_SIG_VAL_FAILED, GetLastError())
 ;
      return false ;
    }
    // get a certificate store for this data
    hStoreMsg = CertOpenStore( CERT_STORE_PROV_MSG
                             , X509_ASN_ENCODING | PKCS_7_ASN_ENCODING
                             , 0            // default cryptographic pro
vider
                             , CERT_STORE_NO_CRYPT_RELEASE_FLAG
                             , hCrMsg
                             ) ;
    if ( hStoreMsg == 0)  {
      LogSecError( "%u %d", ERROR_SECURE_CERTOPENSTORE, GetLastError())
;
      return false ;
    }
    // validate each certificate
    for ( ; pNodeCert != 0 ; pNodeCert = pNodeCert->m_NextSibling)  {
      if ( IsCertValidInAnyStore( pNodeCert->m_RawData
                                , pNodeCert->m_Length + pNodeCert->m_Len
Hdr
                                , hStoreMsg
                                ) == false)  {
        LogSecError( "%u", ERROR_SECURE_CERT_INV_IN_ALL_STORES) ;
```

```
        return false ;
      }
    }  // loop once for each certificate
    // successful validation
    fValid = true ;
  }  // __try
  __finally  {
    // release resources
    if ( hStoreMsg != 0)
      CertCloseStore( hStoreMsg, 0) ;
    if ( pNodeTop != 0)
      delete pNodeTop ;
    if ( hCrMsg != 0)
      CryptMsgClose( hCrMsg) ;
    if ( pbDigestComputed != 0)
      HeapFree( GetProcessHeap(), 0, pbDigestComputed) ;
    if ( pFileData != 0)
      UnmapViewOfFile( pFileData) ;
    if ( hMap != 0)
      CloseHandle( hMap) ;
    if ( hFile != INVALID_HANDLE_VALUE)
      CloseHandle( hFile) ;
    // ensure error logging is enabled in case other functions call it
    EnableSecureErrorLogging( SAPI_LOG_TVDEBUG) ;
  }  // __finally
  // only a successful validation reaches here
  return fValid ;                              // return to the caller
}  // IsPEHFileValidEx
/* function IsPEFileValidEx:
   MSH extension to allow for a WinVerifyTrust alternative
*/
bool WINAPI IsPEFileValidEx( const char *pszFileName, DWORD dwLogFlag, D
WORD dwOptFlags)
{
 bool fValid = false ;                       // assume the worst
 HANDLE hFile = INVALID_HANDLE_VALUE;
    // open the file
    hFile = CreateFile( pszFileName
                      , GENERIC_READ
                      , FILE_SHARE_READ | FILE_SHARE_WRITE
                      , 0                       // no security
                      , OPEN_EXISTING       // the file must exist
                      , FILE_ATTRIBUTE_NORMAL
                      , 0
                      ) ;                       // no template
 fValid = IsPEHFileValidEx(hFile, dwLogFlag, dwOptFlags);
 // set error logging flags for this thread
 EnableSecureErrorLogging( dwLogFlag) ;
 if ( fValid == false)  {
  LogSecError( "%u %s", ERROR_SECURE_FAILED_VALIDATION, pszFileName) ;
 }
    // ensure error logging is enabled in case other functions call it
    EnableSecureErrorLogging( SAPI_LOG_TVDEBUG) ;
 return fValid ;
}  // IsPEFileValidEx
/* function IsPEFileValidExW:
   MSH extension to allow for a WinVerifyTrust alternative
*/
bool WINAPI IsPEFileValidExW( const WCHAR *pszFileName, DWORD dwLogFlag,
 DWORD dwOptFlags)
{
 bool fValid = false ;                       // assume the worst
 HANDLE hFile = INVALID_HANDLE_VALUE;
    // open the file
    hFile = CreateFileW( pszFileName
                      , GENERIC_READ
                      , FILE_SHARE_READ | FILE_SHARE_WRITE
                      , 0                       // no security
```

```
                      , OPEN_EXISTING       // the file must exist
                      , FILE_ATTRIBUTE_NORMAL
                      , 0
                      ) ;                   // no template
 fValid = IsPEHFileValidEx(hFile, dwLogFlag, dwOptFlags);
 // set error logging flags for this thread
 EnableSecureErrorLogging( dwLogFlag) ;
 if ( fValid == false)  {
  LogSecError( "%u %S", ERROR_SECURE_FAILED_VALIDATION, pszFileName) ;
 }
    // ensure error logging is enabled in case other functions call it
    EnableSecureErrorLogging( SAPI_LOG_TVDEBUG) ;
 return fValid ;
}  // IsPEFileValidExW
// All functions below are private to this file
/* function ComputeMessageDigest:
   Hash the PE file contents just like WinVerifyTrust does
   The caller must call HeapFree to free the returned buffer.
   The Crypto APIs used here are in ADVAPI32, available even in Win95
    OSR2.  If we don't like calling them, we can instead use the source
    code in MD5C.c (search the source tree).
*/
static PBYTE ComputeMessageDigest( const BYTE * pbFile)
{
  // get the ranges to hash
  // We assume the file structure has already been validated, so that
  //  these accesses will not fault.
  PIMAGE_DOS_HEADER pHdrDOS = PIMAGE_DOS_HEADER( pbFile) ;
  PIMAGE_NT_HEADERS32 pHdrNT
    = PIMAGE_NT_HEADERS32( pbFile + pHdrDOS->e_lfanew) ;
  PIMAGE_DATA_DIRECTORY pImCert
    = &pHdrNT->OptionalHeader.DataDirectory[ IMAGE_DIRECTORY_ENTRY_SECUR
ITY] ;
  DWORD dwS1 = 0
      , dwE1 = PBYTE( &pHdrNT->OptionalHeader.CheckSum) - pbFile
      , dwS2 = dwE1 + sizeof( DWORD)
      , dwE2 = PBYTE( &pHdrNT->OptionalHeader.DataDirectory[ IMAGE_DIREC
TORY_ENTRY_SECURITY])
               - pbFile
      , dwS3 = PBYTE( &pHdrNT->OptionalHeader.DataDirectory[ IMAGE_DIREC
TORY_ENTRY_SECURITY + 1])
               - pbFile
      , dwE3 = pImCert->VirtualAddress
      ;
  // hash the file's contents
  HCRYPTPROV hProv = 0 ;
  HCRYPTHASH hHash = 0 ;
  PBYTE pbDigest ;
  __try  {                                  // __try / __finally
    // initialize
    if ( CryptAcquireContext( &hProv
                            , 0
                            , 0
                            , PROV_RSA_FULL
                            , CRYPT_VERIFYCONTEXT
                            ) == FALSE)  {
      LogSecError( "%u %X", ERROR_SECURE_CRYPT_ACQUIRE_CONTEXT, GetLastE
rror()) ;
      return 0 ;
    }  // if CryptAcquireContext failed
    if ( CryptCreateHash( hProv             // default provider
                        , CALG_MD5
                        , 0                 // non-keyed algorithm
                        , 0                 // dwFlags, must be zero
                        , &hHash            // handle goes here
                        ) == FALSE)  {
      LogSecError( "%u %X", ERROR_SECURE_CRYPT_CREATE_HASH, GetLastError
()) ;
```

```
      return 0 ;
    }  // if CryptCreateHash failed
    // add the three data pieces
    if ( CryptHashData( hHash, pbFile + dwS1, dwE1 - dwS1, 0) == FALSE)
 {
      LogSecError( "%u %X", ERROR_SECURE_CRYPT_HASH_DATA_1, GetLastError
()) ;
      return 0 ;
    }
    if ( CryptHashData( hHash, pbFile + dwS2, dwE2 - dwS2, 0) == FALSE)
 {
      LogSecError( "%u %X", ERROR_SECURE_CRYPT_HASH_DATA_2, GetLastError
()) ;
      return 0 ;
    }
    if ( CryptHashData( hHash, pbFile + dwS3, dwE3 - dwS3, 0) == FALSE)
 {
      LogSecError( "%u %X", ERROR_SECURE_CRYPT_HASH_DATA_3, GetLastError
()) ;
      return 0 ;
    }
    // get the hash result
    DWORD dwHashSize
        , dwDataLength = sizeof( DWORD)
        ;
    if ( CryptGetHashParam( hHash
                          , HP_HASHSIZE
                          , PBYTE( &dwHashSize)
                          , &dwDataLength
                          , 0
                          ) == FALSE)  {
      LogSecError( "%u %X", ERROR_SECURE_CRYPT_GET_HASH_PARAM_1, GetLast
Error()) ;
      return 0 ;
    }  // if CryptGetHashParam failed
    pbDigest = PBYTE( HeapAlloc( GetProcessHeap(), 0, dwHashSize)) ;
    if ( pbDigest == 0)  {
      LogSecError( "%u %X", ERROR_SECURE_CRYPT_HEAP_ALLOC, dwHashSize) ;
      return 0 ;
    }
    dwDataLength = dwHashSize ;
    if ( CryptGetHashParam( hHash
                          , HP_HASHVAL
                          , pbDigest
                          , &dwDataLength
                          , 0
                          ) == FALSE)  {
      LogSecError( "%u %X", ERROR_SECURE_CRYPT_GET_HASH_PARAM_2, GetLast
Error()) ;
      HeapFree( GetProcessHeap(), 0, pbDigest) ;
      return 0 ;
    }  // if CryptGetHashParam failed
  }  // __try
  __finally  {
    if ( hHash != 0)
      CryptDestroyHash( hHash) ;
    if ( hProv != 0)
      CryptReleaseContext( hProv, 0) ;
  }  // __finally
  // successful return
  return pbDigest ;
}  // ComputeMessageDigest
/* function GetSignature:
   Find the signature in a module's memory image
*/
static bool GetSignature( const BYTE * pbMod, CASN1BERNode * pNodeTop)
{
  // guard the code in case a bogus memory reference causes an access
```

```
  //  exception
  __try  {
    // find where the signature is in the module file
    PIMAGE_DOS_HEADER pHdrDOS = PIMAGE_DOS_HEADER( pbMod) ;
    if ( pHdrDOS->e_magic != 'ZM')
      return false ;                          // error return
    PIMAGE_NT_HEADERS32 pHdrNT
      = PIMAGE_NT_HEADERS32( pbMod + pHdrDOS->e_lfanew) ;
    if ( pHdrNT->Signature != 'EP')
      return false ;                          // error return
    if ( pHdrNT->FileHeader.Machine != IMAGE_FILE_MACHINE_I386)
      return false ;                          // error return
    if ( ( pHdrNT->FileHeader.Characteristics & IMAGE_FILE_EXECUTABLE_IM
AGE)
        == 0)  {
      return false ;                          // error return
    }
    if ( pHdrNT->OptionalHeader.Magic != IMAGE_NT_OPTIONAL_HDR32_MAGIC)
      return false ;                          // error return
    PIMAGE_DATA_DIRECTORY pImCert
      = &pHdrNT->OptionalHeader.DataDirectory[ IMAGE_DIRECTORY_ENTRY_SEC
URITY] ;
    // point to the signature data
    LPWIN_CERTIFICATE pCert
      = LPWIN_CERTIFICATE( pbMod + pImCert->VirtualAddress) ;
    // validate the size field
    if ( pCert->dwLength != pImCert->Size)
      return false ;                          // error return
    // validate the certificate type, since our direct validation
    //  assumes PKCS-7
    if ( pCert->wCertificateType != WIN_CERT_TYPE_PKCS_SIGNED_DATA)
      return false ;
    // parse the ASN.1 BER structure to simplify subsequent navigation
    //  and to partially validate it
    // The length in the WIN_CERTIFICATE structure appears to be
    //  rounded up to a multiple of 8.
    PBYTE pbSig = &pCert->bCertificate[ 0] ;
    LONG lBytesProcessed
      = OneASN1BERLevel( pNodeTop
                       , pbSig
                       , pCert->dwLength
                         - FIELD_OFFSET( WIN_CERTIFICATE, bCertificate)
                       ) ;
    if ( RoundUp( lBytesProcessed
                  + FIELD_OFFSET( WIN_CERTIFICATE, bCertificate)
                , 8
                ) != LONG( pCert->dwLength))  {// if the lengths are not
 right
      return false ;
    }
    // successful return
    return true ;
  }  // __try
  __except( DefaultExceptionFilterEx( GetExceptionInformation()))  {
    LogSecError( "%u", ERROR_SECURE_GET_SIG_FAULT) ;
    return false ;                            // error return
  }  // __except
}  // GetSignature
/* function IsThisCertSignedByAFriend:
   Is this certificate's signer known to us
   Our technique works with certificates for both Zone and AT&T.  Both
    were issued by Verisign.  We could be even stricter, by validating
    the order of the subject fields, since these agree in the two
    certificates and may be standard in certificates issued by Verisign.
   The __try / __except block catches any wildly different structures,
    but developers may complain if this exception drops them into the
    debugger.  So we will attempt to avoid known problems, provided
    this does not overly clutter up the code.
```

```
   Before the list of valid signers grows too big, we hope to
    migrate to Verisign's Authenticated Content Signing technology,
    which will allow us to test for a Zone Labs Publisher ID
    certificate instead of each OEM's name.
*/
static bool IsThisCertSignedByAFriend( const CASN1BERNode * pNodeCert)
{
  CASN1BERNode * pNodeOrga ;
  __try  {                                    // __try / __except
    pNodeOrga = pNodeCert->m_FirstChild ;
    if ( pNodeOrga->m_NbrChildren != 8)      // if not the same as a Zone
 cert
      return false ;                          // we are done
    pNodeOrga = pNodeOrga->m_FirstChild ;    // version
    pNodeOrga = pNodeOrga->m_NextSibling ;   // serial number
    pNodeOrga = pNodeOrga->m_NextSibling ;   // issuer's signature algori
thm
    pNodeOrga = pNodeOrga->m_NextSibling ;   // issuer
    pNodeOrga = pNodeOrga->m_NextSibling ;   // validity
    pNodeOrga = pNodeOrga->m_NextSibling ;   // subject
    pNodeOrga = pNodeOrga->m_FirstChild ;    // first subject field
    for ( ; pNodeOrga != 0 ; pNodeOrga = pNodeOrga->m_NextSibling)  {
      CASN1BERNode * pNodeFind ;
      pNodeFind = pNodeOrga->m_FirstChild ; // sequence
      pNodeFind = pNodeFind->m_FirstChild ; // OID
      if ( IsOID( pNodeFind, bOID_ORGANIZATION_NAME))  {
        pNodeOrga = pNodeFind->m_NextSibling ;
        break ;
      }  // if we found the organization name
    }  // search the siblings
    if ( pNodeOrga == 0)
      return false ;
    // validate the organization name we found
    int ndxValidSigner ;
    for ( ndxValidSigner = 0
        ; ndxValidSigner < nbrValidSigners
        ; ndxValidSigner++
        )  {
      DWORD dwSizeStr = strlen( pszValidSigners[ ndxValidSigner]) ;
      if ( pNodeOrga->m_Length != dwSizeStr)
        continue ;
      if ( memcmp( pNodeOrga->m_RawData
                 , pszValidSigners[ ndxValidSigner]
                 , dwSizeStr
                 ) == 0)  {
        break ;                               // stop looking if we found
a match
      }  // if we found a match
    }  // loop until we find a valid signer
    if ( ndxValidSigner >= nbrValidSigners)  {// if no valid signer foun
d
      for ( ndxValidSigner = 0
          ; ndxValidSigner < nbrValidUniSigners
          ; ndxValidSigner++
          )  {
        DWORD dwSizeStr = wcslen( pwszValidSigners[ ndxValidSigner]) ;
        if ( pNodeOrga->m_Length != dwSizeStr * sizeof( WCHAR))
          continue ;
        if ( memcmp( pNodeOrga->m_RawData
                   , pwszValidSigners[ ndxValidSigner]
                   , pNodeOrga->m_Length
                   ) == 0)  {
          break ;                             // stop looking if we found
a match
        }  // if we found a match
      }  // loop until we find a valid signer
      if ( ndxValidSigner >= nbrValidUniSigners)// if no valid signer fo
und
```

```
        return false ;                          // error return
    }  // if no valid ASCII signers
  }  // __try
  __except( DefaultExceptionFilterEx( GetExceptionInformation()))  {
    LogSecError( "%u", ERROR_SECURE_PARSE_CERT_2) ;
    return false ;
  }
  // successful return
  return true ;
}  // IsThisCertSignedByAFriend
/* function newCASN1BERNode:
   Construct a new node
   This hokey function avoids compiler error C2712 when we compile with
    the -GX option.  We don't need -GX currently (Oct2002), but perhaps
    we will in the future.
*/
static CASN1BERNode * newCASN1BERNode()
{
  return new CASN1BERNode ;
}  // newCASN1BERNode
/* function OneASN1BERLevel:
   Format one level of the ASN.1 BER data
   This function calls itself recursively.
*/
static LONG OneASN1BERLevel( CASN1BERNode * pNodeThis
                           , const BYTE * pbStart
                           , LONG lBytesLeft
                           )
{
  // decide how far to go
  const BYTE * pbEnd = pbStart + lBytesLeft ;
  // guard this code, in case we fall off the end
  PBYTE pbData = PBYTE( pbStart) ;
  while ( pbData < pbEnd)  {
    __try  {
      // quit if no more data
      if ( pNodeThis->m_Parent == 0  &&  *pbData == 0)
        return LONG( pbData - pbStart) ;    // number of bytes processed
      // create a sibling node if this is not the first iteration
      if ( pNodeThis->m_RawData != 0)  {
        // We chain the node as soon as possible, to ensure it is found
        //  during the destruction loop.
        CASN1BERNode * pNodeSibling = newCASN1BERNode() ;
        if ( pNodeSibling == 0)
          return 0 ;                        // error return
        pNodeSibling->m_Parent = pNodeThis->m_Parent ;
        pNodeSibling->m_Parent->m_NbrChildren++ ;// count another child
        pNodeThis->m_NextSibling = pNodeSibling ;
        pNodeThis = pNodeSibling ;          // age the pointer
      }  // if not the first iteration
      // point to the tag, will overwrite for a primitive field
      pNodeThis->m_RawData = pbData ;
      // get the low tag byte
      BYTE bData = *pbData++ ;
      pNodeThis->m_RawTag = bData ;
      bool fPrimitive = ( bData & 0x20) == 0 ;
      bool fUniversal = ( bData & 0xc0) == 0 ;
      // get the tag
      DWORD dwTag = bData & 0x1f ;
      if ( dwTag == 31)  {                  // if a high tag number
        dwTag = 0 ;                         // reset the tag value
        do  {
          bData = *pbData++ ;               // get next tag byte
          dwTag = ( dwTag * 128) + ( bData & 0x7f) ;// ignore overflow f
or now
        } while ( ( bData & 0x80) != 0) ;
      }  // if a high tag number
      pNodeThis->m_Tag = dwTag ;
```

```
      // get the length, if this is fixed length
      // Open: Add support for variable length fields.
      bData = *pbData++ ;                    // get first length byte
      DWORD dwLength = bData ;
      if ( dwLength > 127)  {                // if long form
        DWORD nbrBytes = dwLength - 128 ;
        dwLength = 0 ;                       // reset the length
        DWORD ndxByte ;
        for ( ndxByte = 0 ; ndxByte < nbrBytes ; ndxByte++)  {
          bData = *pbData++ ;                // get next length byte
          dwLength = ( dwLength * 256) + bData ;// ignore overflow for n
ow
        }  // loop once for each length byte
      }  // if long form
      pNodeThis->m_Length = dwLength ;
      pNodeThis->m_LenHdr = pbData - pNodeThis->m_RawData ;
      // call ourselves recursively if a constructed type
      if ( fPrimitive == false)  {
        // describe this tag
        if ( DWORD( pbEnd - pbData) < dwLength)
          return 0 ;                         // error if length is bogus
        // create a child node
        // We chain the node as soon as possible, to ensure it is found
        //  during the destruction loop.
        CASN1BERNode * pNodeChild = newCASN1BERNode() ;
        if ( pNodeChild == 0)
          return 0 ;
        pNodeThis->m_NbrChildren = 1 ;
        pNodeChild->m_Parent = pNodeThis ;
        pNodeThis->m_FirstChild = pNodeChild ;
        // process the child node
        DWORD dwBytesProcessed
          = OneASN1BERLevel( pNodeChild, pbData, dwLength) ;
#if 0
        // 24Dec2003: CA's certificate shows a data type other than
        //  ASN_TYPE_NULL can have length zero.
        if ( dwBytesProcessed == 0)          // if an error
          return 0 ;                         // propagate the error upwar
ds
#endif
        pbData += dwBytesProcessed ;
      }  // if a constructed type
      else  {                                // if a primitive type
        // report the tag
        if ( DWORD( pbEnd - pbData) < dwLength)
          return 0 ;                         // error return if overflow
        // advance past the primitive field
        pNodeThis->m_RawData = pbData ;
        PBYTE pbField = pbData ;             // in case we can format the
 data
        pbData += dwLength ;
        // special handling for the null type
        if ( fUniversal  &&  dwTag == ASN_TYPE_NULL)
          continue ;
        // special handling for an OID field
        if ( fUniversal  &&  dwTag == ASN_TYPE_OID)  {
          DWORD dwNode = 0 ;
          while ( ++pbField < pbData)  {
            dwNode += *pbField & 0x7f ;      // add the new mod 128 digit
            if ( ( *pbField & 0x80) != 0)    // if more digits
              dwNode *= 128 ;                // shift everything over
            else                             // if the last digit
              dwNode = 0 ;                   // reset the total
          }  // loop for all remaining OID bytes
          if ( dwNode != 0)                  // if an incomplete OID
            return 0 ;                       // error return
        }  // if an OID field
      }  // if a primitive field
```

```
    }  // __try
    __except( DefaultExceptionFilterEx( GetExceptionInformation()))  {
      // buffer overrun
      return 0 ;
    }  // __except
  }  // loop while input bytes remain
  // successful return
  return LONG( pbData - pbStart) ;          // number of bytes processed
}  // OneASN1BERLevel
/* function TerminateValidatePEFile:
   Static object destruction
   Avoid memory leaks.
*/
static void __cdecl TerminateValidatePEFile()
{
  // nothing to do yet
}  // TerminateValidatePEFile
/* function CASN1BERNode::CASN1BERNode:
   Constructor
*/
CASN1BERNode::CASN1BERNode()
: m_NextSibling( 0)
, m_FirstChild( 0)
, m_Parent( 0)
, m_Length( 0)
, m_LenHdr( 0)
, m_RawData( 0)
, m_Tag( 0)
, m_RawTag( 0)
, m_NbrChildren( 0)
{
}  // CASN1BERNode::CASN1BERNode
/* function CASN1BERNode::~CASN1BERNode:
   Destructor
*/
CASN1BERNode::~CASN1BERNode()
{
  // delete children first, siblings second
  if ( m_FirstChild != 0)
    delete m_FirstChild ;
  if ( m_NextSibling != 0)
    delete m_NextSibling ;
}  // CASN1BERNode::~CASN1BERNode
```